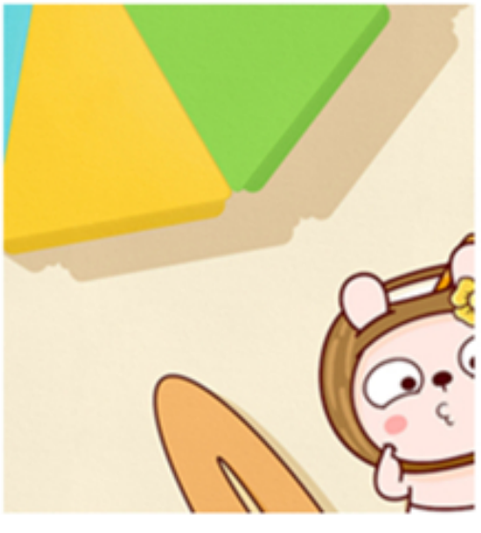
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

HZ BOOKS 华章IT

具有全球影响力的EOS社区引力区联合创始人撰写，EOS官方原产品副总裁等多位专家力荐

从EOS的概况、优势技术、源码实现、原生功能、智能合约、DApp、侧链等7个维度详细讲解EOS的原理和开发实践，已辅导超过20万EOS社区成员

区块技术丛

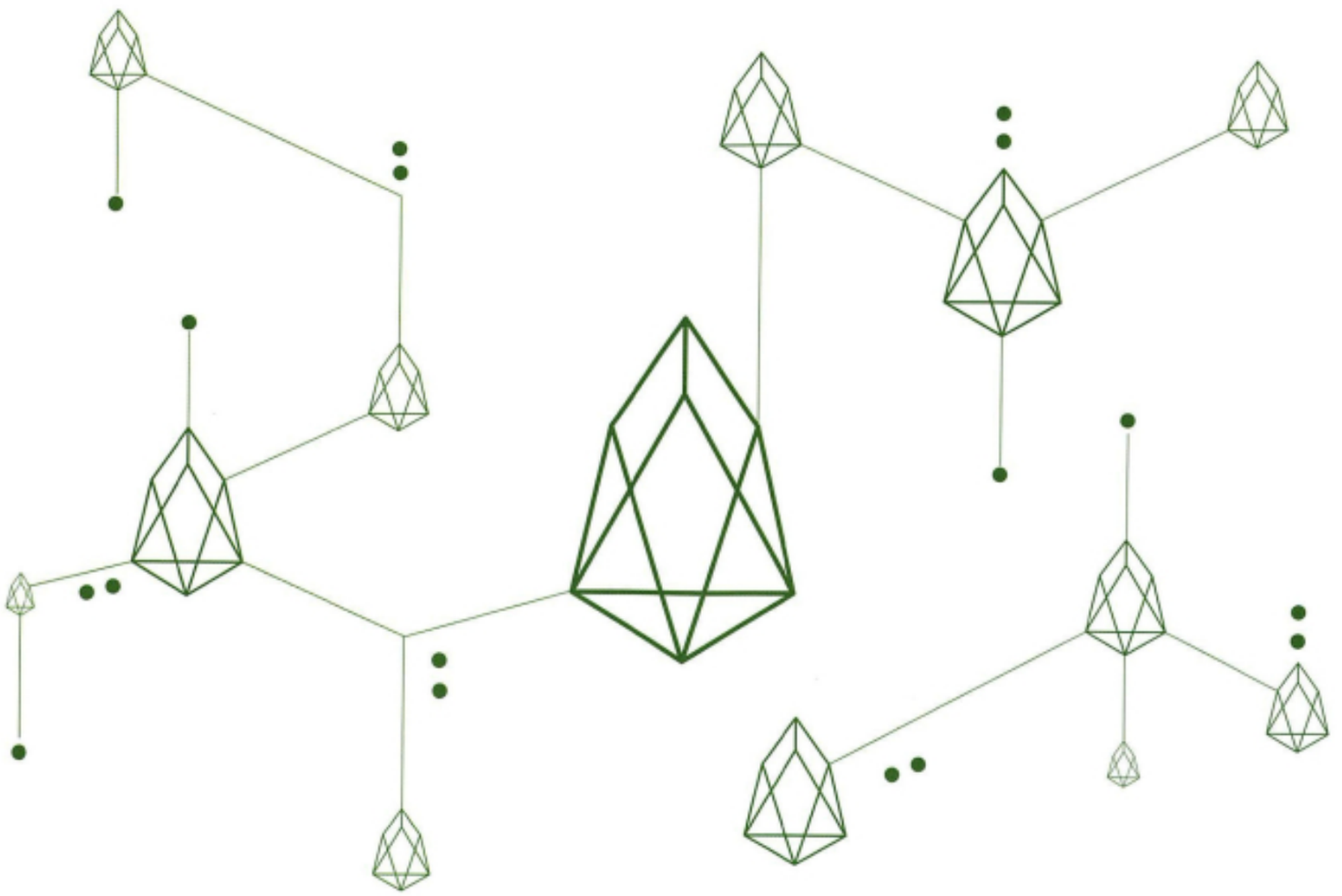
DIVE INTO EOS

Theory Analysis and Developmental Practices

深入理解EOS

原理解析与开发实战

李万才 林琪钧 郭琦康 廖洋阳 ◎著



机械工业出版社
China Machine Press

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

内容简介

这是一部从源码层面剖析 EOS 原理和全方位讲解 EOS 开发实战的著作。

作者是 EOS 全球生态系统中具有全球影响力的 EOS 社区引力区的联合创始人和核心团队成员。本书是引力区在 EOS 主网上的研究和实践成果，得到了 EOS 官方原产品副总裁等多位 EOS 生态领袖的一致好评和强烈推荐。

全书一共 7 章，从 EOS 的概况、优势技术、源码实现、原生功能、智能合约、基于 EOS 的 DApp 和侧链等 7 个方面，详细讲解了从系统原理到软件部署，再到开发的全部细节，适合初学者和具有一定开发基础的读者阅读。

第一部分 原理篇（第 1~3 章）

首先，从 EOS 项目诞生的初衷、愿景、发展历程和版本更迭等角度宏观介绍了 EOS 项目本身；然后，从 EOS 的共识机制、账户体系、网络资源 Token 化、节点治理、脚本与虚拟机等角度展现了 EOS 的技术优势；最后，从源码角度分析了 EOS 项目的整体架构、共识算法的代码实现、网络层的代码实现、区块打包与广播的代码实现，以及关键工具的代码实现。

第二部分 实战篇（第 4~7 章）

这部分主要从实战开发的角度详细讲解了 EOS 网络的启动、原生功能的使用、智能合约的开发、DApp 的开发、基于 EOS 的侧链和竞争链的开发等内容。不仅详细讲解了各项开发所涉及的技术点、开发方法、开发技巧，还包含大量的实战案例。

附录部分展望了 EOS 主网优化的方向，涉及百万 TPS 目标如何实现和跨链交互等方面的内容。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

华章IT | Information Technology | HZBOOKS

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

区块链
技术丛书

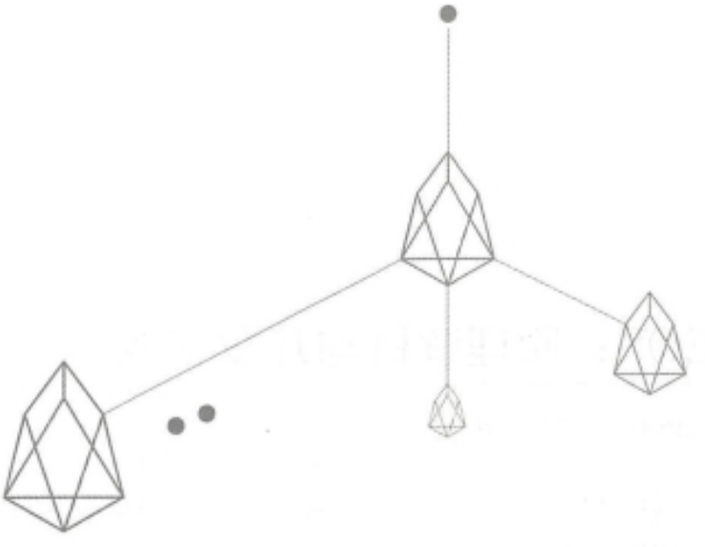
DIVE INTO EOS

Theory Analysis and Developmental Practices

深入理解EOS

原理解析与开发实战

李万才 林琪钧 郭琦康 廖洋阳◎著



机械工业出版社
China Machine Press

仅供非商业用途或交流学习使用

版权相关注意事项：

1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

深入理解 EOS：原理解析与开发实战 / 李万才等著 . —北京：机械工业出版社，2019.1
（区块链技术丛书）

ISBN 978-7-111-61529-3

I. 深… II. 李… III. 电子商务 - 支付方式 - 程序设计 IV. ① F713.361.3 ② TP311.1

中国版本图书馆 CIP 数据核字（2018）第 279525 号

深入理解 EOS：原理解析与开发实战

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：孙海亮　　责任校对：殷　虹
印　　刷：中国电影出版社印刷厂　　版　　次：2019 年 1 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：16.75
书　　号：ISBN 978-7-111-61529-3　　定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Forword 推　荐　序

关于 EOS，总有些狂想是可以实现的（后附英文原版）

托马斯 Block.One 原产品副总裁

热爱 EOS 的人至今已经有数千万了，每个人都有自己的理由。EOS 推崇者和爱好者们，可能来自不同的国家，拥有不同的经历，职业并不一定相同，甚至所用的语言也完全不同，他们却拥有同样的坚持和热爱，怎么看这都是一件神奇的事儿。EOS 正在用自己的魅力感染着更多人了解自己进而了解什么是区块链。

对于 EOS 相关信息的研究及传播，EOS 的爱好者、研究者、BP 候选人都尽了自己最大的努力。

相信本书的读者应该有一部分阅读过我的《 EOS.IO 公约草案 0.1 版本》，其中我

提到一个非常重要的原则“仅限英语”，因为直到现在我依然认为多语种会带来理解上的偏差。当然，其中还有一部分原因是我对其他语言的掌控能力可能不如英文。所以在李万才博士告诉我，他的这本书写完了，并希望我阅读之后撰写一篇序言时，我的第一反应是“我应该怎么去读它”以及“我是否会对他的著作内容产生理解偏差”。

中文并不是一门可以快速从入门到精通的语言，它的复杂程度是难以用语言进行描述的。还好李万才博士的翻译团队用他们的专业和耐心辅助我读完了这本书。

从内容上说，这是一本对 EOS 爱好者甚至区块链观察者非常友好的书。它是一本专业的书籍没错，但它在最大程度上做到了用最通俗的语言去完成 EOS 系统原理的文字表述，哪怕是对 EOS 了解并不深刻的读者，也能够看懂这本书在说什么。在实际操作层面，本书中基于 EOS 的 DApp 开发和基于 EOS 的侧链 / 竞争链开发，也做到了尽

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

可能详细，它像是一本产品说明书一样，让初学者也能够尽快上手。

我相信李万才博士在写这本书时，一定倾注了自己对 EOS 强大的热爱，否则它不会同时具备“通俗易懂”和“内容深刻”两个特征。如果说原理篇是新手入门的基础、实战篇是手把手教大家开发的教程说明，那么附录部分，则是为读者完成认知升级的含金量较高的一个部分。李万才博士在附录中所展示的内容并不是一些专属名词或者参考网站，而是 EOS 主网的系统优化方向。这不是刚接触 EOS 的人能够快速抵达的思想高度，也并不是每一个开发 EOS 的人都能拥有的思想深度。所以，这虽然是一本初学者都能看懂的书，但在对 EOS 的认知和理解上，李万才博士确实是深刻而富有有远见的。从他对 EOS 原理介绍时所用的激情满满的文字中，你可以看到他对 EOS 的执着。如果你也深切地热爱 EOS，并坚信 EOS 的未来无限美好，那么，读这本书时你所获得的灵魂共鸣不会比我少。

至于什么人需要阅读这本书，我想远远不止 EOS 的爱好者、研究者和开发者，还应该包括区块链的爱好者、研究者、开发者和观察者。当然，还应该包括对新生事物充满好奇的人们。

EOS 每天都在不断发展壮大，每一个脚印都伴随着惊叹。它诞生在聚光灯之下，一举一动都那样引人注目，中文名的挑选轻而易举地使它成为区块链行业的头条新闻，主网相关的资讯都能在短短几小时内传播到世界各地——对 EOS 的热爱和关注，可以打破时间和地域的格局，这是一件多么伟大的事情。何况即便是到了已经被大众所认知的今天，EOS 也以自己与众不同方式在生长，逐渐从小众的狂热变成了大众的认可。

今天如果还有人问区块链是否可以落地、有什么已经落地的成果，那么 EOS 和 DApp 必定是答案之一，这也是我在序言标题中所说的“关于 EOS，总有些狂想可以

实现”。EOS 三个字母、BM、千万人的共同狂热，这些词汇聚在一起，是这么美妙而令人激动。

每个人都有自己的技术信仰，至于你的信仰是否能够成真，只看你愿意为这个技术付出多少贡献，以及这个技术能被拥有信仰的人们推进到怎样的地步。EOS 是我和李万才博士，还有众多的 EOS 爱好者们共同的、不可动摇的信仰。希望这本书能够为读到这篇序言的你带来一些帮助，也很高兴能够与李万才博士一起为 EOS 的推广做出一些微薄的贡献。关于 EOS，未来已来，梦想可期。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Foreword Amazing EOS idea is coming soon

Thomas B. Cox Former VP of Product of Block.One

There are tens of millions of EOS enthusiast,and everyone has their own reasons. It's amazing that they come from different countries, have different experiences and occupations, and speak different languages, yet share the same persistence and passion. EOS is so attractive that more people start to understand EOS and what blockchain is.

EOS enthusiasts, researchers, and block producers and candidates have done their best to research and disseminate information related to EOS.

I believe that some readers have read the early versions of the EOS.IO constitution that I drafted, in which I mentioned a very important principle "English only", and I still believe that mixing multiple languages will bring misunderstanding. Of course, part of the reason is that I'm not as good at other languages as I am at English. So when Dr. Li Wancai told me that his book was finished and hope i can write a preface after reading, my first reaction was "how should I read it" and "how can I understand the content of his book".

Chinese is not a language that can be quickly learned and easily mastered. Its complexity is difficult to describe. Fortunately, Dr. Li ' s translation team brought great professionalism and patience.

In terms of content, I believe you will find this is a very friendly book for EOS enthusiasts and even blockchain watchers. It is a professional book, using the most popular language to express the principles of the EOS system. Even readers who don't know

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

much about EOS can understand what the book is saying. In terms of actual operation, the development of EOS-based Dapp and EOS-based side chain/competition chain in this book are as detailed as PoSsible, just like a product manual, which enables beginners to get started as soon as PoSsible.

The book reflects the time and effort Dr. Li Wancai put into writing. How else could he achieve "easy to understand" and "deep content" at the same time? If the principle article is the basis for beginners and the actual practice is the development tutorial section, then the appendix is the high value reference section. What Dr. Li Wancai shows in the appendix is not some exclusive terms or reference websites, but the direction of system optimization of EOS main network. People who are new to EOS are not able to reach the level of thought quickly, and not everyone who develops with EOS can have the depth of thought. Therefore, although this is a book that beginners can understand, Dr. Li Wancai is indeed profound and far-sighted in his cognition and understanding of EOS. From his passionate introduction to EOS principles, you can see his dedication to EOS. If you are deeply passionate about EOS and believe that the future of EOS is infinitely better, then you will gain a lot of soul resonance when reading this book;

For anyone who needs to read this book, I think far more than EOS enthusiasts, researchers and developers. It should also include blockchain enthusiasts,developers and observers,of course,include people who are curious about new things.

EOS continues to grow and develop every day, and every footprint is accompanied by wonder. EOS was born in the spotlight, every movements are so eye-catching.The selection of Chinese names has easily become the headline news of the blockchain industry. All the information related to the mainnet can be spread to the world in just a few hours. - the attention to EOS can break the pattern of time and Region,which is a great thing. Moreover, even today, when it has been recognized by the public, EOS is growing in a different way, and gradually becomes a public recognition from the fanaticism of the niche.

Today, if someone asks whether blockchain can landed and what has already been done, then EOS and the eos based dApp must be one of the answers, which is also what I said in the preface title "Amazing EOS idea is coming soon". The three letters "EOS",

仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

"Daniel Larimer", "thousands of EOS enthusiasts", it's so wonderful and exciting that these words come together.

Everyone has their own beliefs about the role and purPoSe of technology. But if you want your beliefs to become visible in the world, your success depends on how much you are willing to contribute to the technology and how far it can be advanced by people of faith. Faith is key. And EOS happens to be a belief shared by me and Dr. Wancai li, as well as by EOS enthusiasts. I hope this book can bring you some help in reading this preface, and I'm glad to make some insignificant contributions to the promotion of EOS together with Dr. Li Wancai. Regarding EOS, the future has come and the dream is PoSsible.

前　言 *Preface*

为什么要写这本书

2014 年的夏天，我大部分时间都待在热气腾腾的深圳，身边的矿机嗡嗡作响，环境温度在 40 度以上，可那时我们这些创业者们却好似身在冰窟，心里无比凄凉。那时的我们没见过多少世面，也没经历过太多波折，比特币、莱特币、狗狗币是我们仅有的对区块链这座冰山的小小一角的窥视。但也正是在那个绝望的时刻，我接触到了 Bitshares，这也是 Daniel Larimer 的第一个项目，当然觉得特震撼。在那个时代，出现这个级别的产品，简直就如同比特币最初的横空出世，让人敬畏。

天才总是充满着野心和想象力，之后的 STEEM，现在的 EOS，都是 Daniel Larimer（以下简称 BM）不断为区块链，乃至整个商业世界带来的超级创新项目。特别是 EOS，它勾勒出了区块链 + 云计算 + 大规模商业应用的未来图谱，让所有对区块链心生好感的人都充满了期望：这种技术如果普及了，能从多大程度上改变世界，改变我们自身呢？

为什么我们如此热爱 EOS，并选择 EOS 作为一个智能合约平台的载体呢？我先给出自己的理由：

（1）我喜欢与众不同的人和不随波逐流的人，尤其是还能做出令人觉得瞠目结舌的产品的人。我是 BTS 和 STEEM 的深度用户，我觉得它们优雅、精致、超前于时代；我觉得它们的架构和技术能够实实在在地让我感到既酷又适于每日品玩。我看过 BM 的很多文章，那种既带着些许忧郁感又拥有自己的哲学高度，还能深入到技术细节的文字，让我感触良多。也许，在潜意识中，我也想成为 BM 那样的人吧。

（2）EOS 是一个从实践中、从小而美的应用中、从历经时间磨砺的系统中抽象出来的某种设计理念的实体化，不同于各种无中生有、野心勃勃的大概念和大设想，它更为亲切和熟悉，像是一个被顺水推舟解出的方程式，像是一部美剧中可独立成章的主线剧情，它自然而然地发生，无比贴合你一直以来的期待。

（3）看好 EOS，是一个迭代更新的过程。随着不断地为 EOS 贡献微小的价值，我能感受到价值的积聚，以及这种价值的吸引力。而随之而来的成就感又激励我进一步做出能力所及的更多贡献，在这个过程中，我和 EOS 在某种层面上已经融为一体。我经常笑谈，自己是 EOS 的看门狗，便是此意。

（4）EOS 具有无限性。虽然 EOS 会受到现有硬件和软件架构的限制，但是 EOS 的出现对于创新的激发是无限的。因为它让很多人意识到这个巨大生态的价值，以

及各种商业逻辑迁移的可能性。就我所知，很多针对 EOS 的创新已经在酝酿之中了。我相信，EOS 主网上线之日起的一个月内，会有数以百级的应用展现在我们面前。

面对自己钟爱之物时，我们总想为它赋予更深刻、更高尚的意义，也许我也不自觉地美化了它，但是，我不会觉得惭愧，因为我忠于这种内心的选择，而且不会因此伤害到其他人。

当然，EOS 目前还处在发展初期，还没有那么成熟和完善，我经常会被问到的一个问题：我想学习 EOS 的开发，怎么入门？确实，我们特别需要一个有体系的、专业的能带着程序员、爱好者们真正走入 EOS 这个新的技术平台大门的教材，这也是这套书的最终目的。我们在引力区集合了自己的技术团队和社区的开发者们，通过社区大家长期深入沟通交流，找到了帮助初学者实现入门到精通的学习方法，总结出了我们真正需要的那些知识和需要跨过的那些门槛，然后以体系化的方式整理成册。我们的目标很明确，帮助 EOS 社区培养更多的开发者，也帮助开发者们进入一个超级公链生态。我觉得，公链和人之间是通过这种引力的方式互相链接起来的，这才是区块链的本质。

读者对象

这里根据软件需求划分出一些能使用本书的用户团体：

- 区块链技术初学者
- EOS 技术爱好者
- 基于 EOS 的 DApp 开发者

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

- EOS 生态建设者
- 开设相关课程的大专院校师生

本书特色

本书基于 EOS 引力区的技术团队在 EOS 主网上累积的实战技术开发经验撰写而成，书中介绍了 EOS 项目的技术全貌，内容深入浅出，旨在帮助读者快速理解 EOS 技术亮点以及实现过程。从实际软件开发案例、EOS 原生功能使用、智能合约开发，以及基于 EOS 的 DApp 开发和侧链开发等几个方面，详细介绍了从软件部署到具体开发的全部细节，适合初学者和具有一定开发基础的读者阅读。

如何阅读本书

本书分为两大部分：

第一部分为原理篇，本篇从三方面着手，即括基础介绍、技术优势分析和代码原理分析三部分，由浅入深地介绍了 EOS 系统的原理。

第二部分为实战篇，通过项目实例详细讲解 EOS 主网原生功能使用、智能合约开发、基于 EOS 的 DApp 开发和基于 EOS 的侧链 / 竞争链开发。

其中第二部分通过具体的开发项目，手把手地带初学者快速掌握基于 EOS 的项目开发并掌握 DApp 开发逻辑，建议有一定开发能力的用户直接阅读这部分。对于初学者，则建议从头阅读，在掌握了 EOS 基本技术原理的基础再学习第二部分。

除上述两大部分，本书最后还包含一个附录。附录部分介绍了几个 EOS 主网的系统优化方向。

勘误和支持

除封面署名外，参与及支持本书编写工作的还有 blockone 刘函、eoscannon 施瑞琦、eoshuobipool 闻 华 强、eosnewyork Kevin、eoscanada Richard、liquideos Galian、EOS 开发者社区陈澄、eoslaomao 赵余、eosasia 郭达峰、eosbixin 吴广庚、eosbeijing 李想、eosmeetone 高锋、eosflytomars 龙志峰、starteos 李昂、eosstore 陆海峰、GBAC 郑玉山、比特股理事刘嘉陵、zbeos 游凯强、oraclechain 赵微、eos 硅谷 Bo、eosauthority Roshan。

由于作者水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的

地方，恳请读者批评指正。为此，我特意创建了一个在线支持与应急方案的二级站点 http://book. eosgravity.com/。你可以将书中的错误发布在 Bug 勘误表页面中，同时如果你遇到任何问题，也可以访问 Q&A 页面，我将尽量在线上为读者提供最满意的解答。书中的全部源文件除可以从华章网站[⊖]下载外，还可以从这个网站下载，我也会将相应的功能更新及时发布出来。如果你有更多的宝贵意见，也欢迎发送邮件至邮箱 liwancai@eosgravity.com，期待能够得到你们的真挚反馈。

致谢

这本书的出版发行，得益于众多社区成员、朋友们的帮助。

首先要感谢 EOS 引力区在本书成文过程中提供的帮助。因为有了 EOS 引力区的专业知识、敬业态度及为区块链无私奉献的精神，才有了这本书的顺利问世。

其次也要感谢 GBC 引力区块链、币晓爷、哈希女神、链就社、币乎、币车等媒体、项目方在本书出版发行过程中提供的支持。

同时要感谢薛虹森、李俊、沈雪红、孙振、陈翔、周宇、彭有旺、星辰、仲一、陈强、陈建清、康枝旺、马健宁等各位引力节点们助力 EOS 生态繁荣，以及为本书提供的全方位的反馈。

还要感谢邱琪、何德林、张怡军、刘定强、陈科伟、杨建锋、肖志铭、卜守强、刘晓凯等社区活跃成员一路的陪伴与鼓励。

感谢众多的社区成员给出的宝贵意见，他们直接或间接参与了本书的撰写工作。由于篇幅有限，在此我们无法一一致谢，但我们一直心怀感恩。在区块链探索自由的道路上，感恩一路有你。

感谢张国强老师的引荐，在你的努力下才促成了这本书的合作与出版。

感谢机械工业出版社华章公司的编辑杨福川和孙海亮，在这几个月的时间中始终支持我们的写作，你们的鼓励和帮助引导我们顺利完成全部书稿。

最后感谢我的父母、妻子小红以及女儿西西，你们的陪伴和理解时刻为我灌输着信心和力量！

谨以此书献给我最亲爱的家人，以及众多热爱 EOS 的朋友们！

李万才

⊖ 参见华章网站 www.hzbook.com. ——编辑注

目　录 *Contents*

推荐序
前言

第一部分　原理篇

第 1 章　EOS 基本介绍……2

1.1　区块链 2.0 的痛点……2
1.2　EOS 项目定义与愿景……5
1.3　EOS 项目的发展历程……8
1.4　EOS 项目版本更迭简况与现状…11

第 2 章　EOS 项目技术优势解析……15

2.1　EOS 的共识机制（BFT-DPoS）…16
2.1.1　超级节点竞选……18
2.1.2　持币者投票……19
2.1.3　超级节点治理……20
2.2　EOS 的账户体系……20
2.2.1　基于角色的多层级账户体系…21
2.2.2　账户权限管理……21
2.2.3　强制延迟消息执行策略……22
2.2.4　密钥丢失或被盗后的恢复……23
2.3　网络资源 Token 化……23
2.3.1　带宽和日志存储……23
2.3.2　计算和计算积压（CPU）……24
2.3.3　状态存储器（RAM）……24
2.4　EOS 的节点治理……25
2.4.1　宪法和协议指导下的关键操作……25
2.4.2　宪法和协议的升级……26
2.5　脚本与虚拟机……27

第 3 章　EOS 代码原理分析……29

3.1　EOS 项目代码框架总览……29
3.1.1　脚本层……29
3.1.2　插件层……31
3.1.3　合约层……36
3.1.4　支持层……39
3.2　共识算法代码实现……42

仅供非商业用途或交流学习使用

3.2.1 DPoS 机制实现关键代码分析……43
3.2.2 BFT 算法实现关键代码分析…53
3.3 网络层代码实现……61
3.4 区块打包与广播的代码实现……66
3.5 系统合约层代码实现……70
3.5.1 eosio.system 合约代码分析……71
3.5.2 eosio.Token 合约代码分析……79
3.5.3 eosio.msig 合约代码分析……82
3.5.4 eosiolib 依赖库……84
3.6 EOS 关键工具的代码实现……89
3.6.1 nodeos 代码实现……89
3.6.2 cleos 代码实现……92
3.6.3 keosd 代码实现……94

第二部分 实战篇

第 4 章 EOS 网络启动与原生功能介绍……104
4.1 EOS 项目安装部署与编译……104
4.2 配置文件修改与测试……106
4.3 EOS 测试网启动……108
4.3.1 Bios 节点……109
4.3.2 预出块节点……109
4.3.3 普通节点……111
4.4 EOS 项目常用功能……112
4.4.1 链上信息查询……112
4.4.2 钱包管理……126
4.4.3 账户创建与权限管理……132
4.4.4 代币的交易……135
4.4.5 交易的签名与发送……137
4.4.6 系统资源获取……138
4.5 智能合约的部署与调用……140

第 5 章 智能合约开发……142
5.1 智能合约简介……142
5.1.1 智能合约定义……143
5.1.2 智能合约历史……148
5.1.3 智能合约优劣势……154
5.1.4 智能合约应用场景……155
5.2 EOS 中的智能合约……165
5.3 EOS 智能合约文件结构……169
5.3.1 .abi 文件……170
5.3.2 .wasm/.wast 文件……171
5.3.3 eosioapp 编译器使用……172
5.4 EOS 智能合约依赖库介绍……179
5.5 EOS 智能合约实战项目……196
5.5.1 HelloWorld 合约……196
5.5.2 数据库合约……200
5.6 常见 bug 与 debug 思路总结……205

第 6 章 基于 EOS 的 DApp 开发…207
6.1 什么是 DApp……207
6.1.1 比特币是第一个 DApp……208
6.1.2 DApp 命名的原因……208
6.1.3 DApp 运作的法律模型……209
6.2 开发 DApp 的一般流程……210
6.2.1 建立共识的机制……210

仅供非商业用途或交流学习使用

6.2.2 设计分发 Token 的机制 ····· 211
6.2.3 常见 DApp 开发阶段 ····· 213
6.3 DApp 的应用领域 ····· 214
6.3.1 游戏领域 ····· 214
6.3.2 社交 / 内容分发平台 ····· 216
6.3.3 数字货币钱包 ····· 216
6.4 基于 EOS 的 DApp 开发思路 ····· 217
6.5 DApp 开发实例——网页发币 ····· 227
第 7 章 基于 EOS 的侧链 / 竞争链开发 ····· 232
7.1 侧链 / 竞争链的概念与起源 ····· 232
7.2 侧链实现方式 ····· 235
7.3 主流侧链项目 ····· 239
7.4 EOS 侧链 ····· 242
7.5 EOS 竞争链 ····· 245
附录 A EOS 系统优化方向展望 ····· 248

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

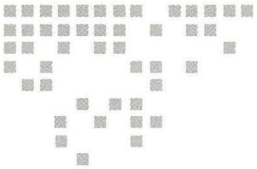
第一部分 Part 1

原 理 篇

- 第 1 章　EOS 基本介绍
- 第 2 章　EOS 项目技术优势解析
- 第 3 章　EOS 代码原理分析

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用



Chapter 1 第 1 章

EOS 基本介绍

1969 年，互联网在美国诞生，这是人类历史上的又一次革命。起初，互联网仅仅被运用在军事方面，后来被应用在生活中的各个方面。也正是它的广泛运用，才使得我们现在的生活如此便捷。

随着互联网的广泛应用，我们正与全世界“分享”着自己的个人信息，导致我们的个人信息经常被泄露，给我们的生活带来很大的困扰。主要的原因是，我们使用的各个平台的应用，其服务都是中心化的。这意味着平台使用的服务器一旦被黑客攻击，我们所有的数据都会被盗取，从而造成信息泄露和财产损失。这些问题的存在使得人们对互联网安全的担忧越来越重，互联网需要针对这些问题给出解决方案。所以，去中心化与保障信息安全的区块链技术应运而生。

1.1 区块链 2.0 的痛点

区块链的总体发展经历了三个阶段：1.0 时代的代表是去中心化账本，2.0 时代的代表是智能合约，3.0 时代的代表是行业应用。

早期区块链技术用途比较单一，仅仅是一种被用来保证去中心化架构下数字货币交易可信性的技术方案中的一部分。但是，区块链技术在运用过程中所体现出来的几点特性，决定了其可以被用在更为广阔的领域：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 保证信息的共享且真实可信：该特征可以被运用在股权众筹领域，使得信息对投融资双方更加对称，记录不会被篡改与删除，也使得股权转让和登记更加安全便捷。
- 交易费用成本较低：该特征可以被运用在证券的交易上，每次记录的数据均不可逆，这样可以避免双方最后确认时产生不必要的分歧。
- 链上信息不可逆：该特征可以被运用在各种彩票行业，防止内部人员进行暗箱操作。
- 用户身份信息匿名化带来的安全性：该特征可以被运用到各个商业性的购物平台，其可保证用户的信息不泄露，也会减少我们生活中被各种电话广告骚扰所带来的困扰。

作为第一个支持智能合约的区块链平台，以太坊被看作区块链 2.0 时代的代表产物，但作为早期技术，它显然还不够成熟，具体表现体现在以下几个方面：

1. 调用智能合约会产生昂贵的费用

目前在以太坊平台上的智能合约每次调用大概需要支付 1～3 美元的费用，这对于一个商业应用来说是一笔不小的成本，这样会把一大批开发人员拒之门外。例如一个 DApp 创业公司，考虑到系统功能上的需求，可以选择在以太坊平台上开发一个应用以很好地利用区块链的优势，但在后续的项目运维过程中，每次对智能合约进行调用都需要支付费用，对于高并发场景，其单位时间内调用次数可以达到上万次，从而带来的成本会高到让人难以接受。

2. 交易处理不公平

交易的执行需要按照智能合约预先确定的算法执行，在执行每个指令时都需要产生一定的消耗，这个消耗在以太坊中是以 Gas 作为基本单位的。不同的指令对 Gas 的消耗也是不同的。简单理解，Gas 就相当于汽油，只要发动汽车，无论车向哪个方向开动，都需要消耗汽油。每笔交易都需要包括一个 Gas limit（有的时候被称为 startGas）的费用。

Gas limit 是单笔交易允许的 Gas 最大可消耗数量，可以理解为交易本身的服务费。而不同的是，这里面会有一笔小费产生，被作为交易手续费支付给矿工。矿工是有权利选择先打包哪一笔交易的，也就是说支付的交易费越多，矿工就会倾向于优先打包

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

那笔交易，交易确认速度也越快。这样就会存在不公平性。

3. 网络堵塞交易速度低

以太坊目前能够支持的每秒处理交易的速度（TPS）非常有限，每秒只能处理20～30笔交易。一个中型的面向消费者的应用就能让整个以太坊网络完全堵塞。以太坊网络的所有节点都在处理相同的智能合约，而这些智能合约的状态就记录在公共区块链的账本上，导致系统效率低下，无法承载上千万的DApp运行。

关注数字货币和区块链的朋友大概都知道，2017年年底一款基于ERC-721的区块链小游戏《撸猫》(CryptoKitties)。在CryptoKitties平台上，猫咪具有生成、繁殖和交易的功能，每只虚拟猫，其实在以太坊区块链平台上都是一条代码，每一条代码都是唯一的。猫咪的买卖、生育和生成等区块链交易行为，在技术上需要支付给以太坊矿工一笔交易费，也就是所谓的Gas，让他们来进行计算验证。而正是由于以太坊共识算法导致的性能不足，导致网络阻塞、交易延迟，进而导致体验很差，甚至使整个以太坊陷入瘫痪。

4. 不支持浮点数与代码的不可修改

作为搭建DApp的平台，支持浮点数据是必然要求，因为浮点数会被用到很多的应用实例中，比如风险建模、科学计算以近似值比准确值更加重要的情况。智能合约在设计时需要考虑的重要问题之一就是可升级性，因为合约升级是必然的。以太坊中的代码是完全不可修改的，由于其采用哈佛计算机结构，故也不可能将代码在内存中加载并执行，代码和数据是被完全分离的。目前只能够通过部署新的合约来达到升级的目的，这可能需要复制原合约中的所有代码，并将老的合约重定向到新的合约地址。给合约打补丁或是部分升级合约代码在EVM中是完全不可能的。

总之，这就是以太坊当前面临的危机。不可否认，以太坊作为第一个区块链虚拟机存在诸多可优化方向，这和绝大多数新生事物一样，这种问题都使得移植主流编程语言的代码变得困难重重。目前以太坊只能支持专门定制的开发语言，这是件令人遗憾的事。为了解决以上问题，Daniel larimer（BM）团队发起了EOS项目。该项目旨在通过性能与功能的提升让区块链的应用超越发币，并可应用到政府、商业等领域，从而打造一个可以用于商业级别开发的DApp平台。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

1.2　EOS 项目定义与愿景

EOS 的全称是 Enterprise Operation System，字面意思是商业分布式系统，即适用于商业分布式应用的区块链操作系统。EOS 作为一条高性能的公链，交易几乎可以在一秒内确认。比特币的诞生就是为了实现一个点对点的加密货币系统，以太坊的出现实现了图灵设备的虚拟机，该机制的成功运行使得区块链能够成为类似于 Windows 的计算平台。EOS 平台可以理解为一个系统，在该系统上的智能合约就类似于操作系统下的各个程序与软件，例如 EOS Token 是其系统智能合约定义的。

以下主要介绍 EOS 系统应用平台的相关特性。

1. 支持数百万级别用户

现在我们每天使用的软件，例如微信、支付宝、淘宝等，其用户量都在亿级别，更高者能达到十亿级别。因此在高峰时期，每个软件承担的通信量可能都在十万或百万级别（支付宝双十一期间交易量最高达到每秒百万）。而 EOS 作为一个区块链系统，后续会有大量的公司、个人在其上部署智能合约、开发 DApp，可想而知，随着生态建设的发展，EOS 的用户量会急剧增长，对其性能要求自然会极高。因此当大量的商业级别的应用使用区块链系统时，需要被选择的区块链系统能够支持数百万级别吞吐量，这恰恰也是 EOS 的优势所在。

2. 免费使用

就目前而言，几乎所有的应用都是基于 Windows 系统与 Linux 系统开发使用的。为了吸引更多的用户使用，绝大多数应用为用户提供了免费的服务，但也会有部分软件由于开发成本过高以及产品定位不同，会根据开发者的定价向用户索要相应金额的费用。目前大部分用户已经习惯了这样的模式，假设淘宝变成了一个付费平台，而其他用户体验不变，那么其付费平台的成本必然会附加到用户的最终消费中去，会间接提高用户的使用成本。因此 EOS 向用户提供免费服务的策略会吸引更多的用户数量。用户不必为了使用平台而付出额外费用。有了足够的用户规模之后，开发者与企业可以根据自己的商业模式来选择赢利的手段。

3. 简单的升级与漏洞修复

举个简单的例子：如果你是一个喜欢游戏的人，相信对《王者荣耀》这款游戏不

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

会陌生，因为它的易上手与快节奏深得大家喜爱。在吸引更多用户使用的同时为了修复已发现的 bug，《王者荣耀》经常会升级与更新，这个流程对于用户并没有太大的影响，就是进行一键升级然后系统会自动加载。但是如果它的升级非常复杂，就一定会导致用户的流失。相同的道理，一个应用平台就相当于一款游戏，而基于此平台开发的应用就是用户。对于应用的开发者来说，简单的升级是很喜爱的。平台在研发出来时不可能十全十美，其存在的 bug 与漏洞可能会影响应用的使用，这就需要此平台不断进行更新与修复。

4. 较低的延迟性

当你遇到网站卡顿、播放视频加载的缓慢或酣畅淋漓打游戏突然掉线时，你一定会很无奈，你一定会选择没有延迟、卡顿，但拥有同等效果的替代品。上述这些问题都会直接影响用户对互联网产品的直观体验，并且会影响用户选择产品的倾向性。网页加载时间过长会直接导致用户流失。据某性能管理平台调查发现，近 50% 的用户最多愿意等 3 秒，如果 3 秒之内无法加载出自己想要的画面，他们就会失去耐心。对于某些购物网站来说，加载时间每延长 1 秒，一年就会减少数以亿计的营收。所以及时的反馈是获得良好用户体验的基础，超过几秒的延迟时间会导致程序应用的竞争力大大降低。

5. 强大的串行性能

对于串行和并行的概念我们可以与物理上面的串行与并行结构进类比。在实际时钟频率比较低的情况下，并行结构可以同时传输若干比特，所以处理速度确实比串行结构快。但是，随着技术的发展，时钟频率越来越高，并行导线之间的相互干扰越来越严重。并行接口因为有多条并行且排列紧密的导线，随着时钟频率提高到一定程度，传输的数据已无法恢复。而串口因为导线少，相互之间干扰容易控制，反而可以通过不断提高时钟频率的方式来提高传输速率，而且串口的端子也会比较小。这就是现在高速传输都采用串行方式的原因。“众人拾柴火焰高”是句老话，但电脑领域却发生了多根线比不过一根线的怪事。

无论从通信速度、造价还是通信质量上来看，现今的串行传输方式都比并行传输方式更胜一筹。但是我们这里需要强大的串行性能与上述的原因并不一样，而是由于一些应用程序的命令执行必须是顺序的，从而无法用并行算法进行实现。诸如交易所

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

之类的应用经常需要处理大量的串行操作，因此一个成功的区块链架构需要具有强大的串行性能。

6. 强大的并行性能

所谓的并行性能就是指多线程的性能。使用多线程的优势在于：可并行处理任务，减少单个任务的等待时间；线程间可共享资源；在多核情况下可充分利用 CPU 资源，发挥多处理器的强大性能，提升资源利用率及系统的吞吐率；提供更好的 GUI 交互体验（如腾讯视频可边下边播）；充分利用服务器硬件资源；多线程并发内存消耗比较少与 CPU 消耗比较小。对于大规模的应用程序，并行方式能够很好地解决大量的数据运算问题，因此一个成功的应用平台也需要支持强大的并行性能。

若 EOS 想要成为一个成功的智能合约的商业应用平台，就需要具备以上六点。

现今，在 EOS 主网已经成功上线的情况下，它采用的 BFT-DPoS(delegated prove of stake) 共识机制，其最终目标是支持每秒处理 1 万～10 万笔交易。同时 EOS 还采用了并行处理的技术，可以使交易规模达到每秒几百万次。这就表示 EOS 同时可以支持上万个分布式应用程序在它的平台上运行、工作；EOS 还提供一套基础功能比较完善的操作系统，使得不同类型的区块链应用都可以使用。对于开发者来说，相比于以太坊系统，在 EOS 操作系统上可以更加迅速地开发自己的应用，而不需要自己再开发操作系统，从而使得区块链应用开发被简化，降低了开发难度；EOS 并不是根据用户提供手续费的数量决定交易执行顺序的，而是采用所有权的模式进行分配，也就是按照用户拥有 EOS 代币的比例来分配其使用 EOS 网络的带宽、存储和运算资源。

假如你拥有百分之十的 EOS 代币，就意味着你拥有百分之十 EOS 的带宽、存储等资源，无论网络上其他的地方是否拥堵，对你都造不成影响。当然这对于代币投资者来说没有太大的意义与作用，因为他们的目的与长处并不是开发。但是这些资源对于在 EOS 平台上开发应用程序的人来说则是非常有价值的。因为他们需要使用这些资源来开发与运行他们的应用程序。EOS 中允许投资者将自己的资源租借给其他开发者，应用程序开发成功了就会引来更多的用户，这个时候投资者手里的这些资源自然变得更有价值。

而对于初期的开发者来说，他们也不需要花大量的钱去买服务器，在这样互利互赢的情况下，EOS 代币的价格必然会提高。不仅对于投资者来说能得利，对于初期的

非卖品，仅供非商业用途或交流学习使用

创业公司来说也会大大降低创业成本，他们只需要有好的点子，写出好的程序，而不需要花太多精力在服务器、存储、带宽等基础设施上。综合以上观点，相信 EOS 会成为区块链世界里的 Windows 或 Linux。

1.3 EOS 项目的发展历程

区块链（Blockchain）技术起源于 2008 年推出的比特币，自那时以来，企业家和开发人员一直在努力推广该技术，以便在单个区块链平台上支持更广泛的应用。虽然一些通用区块链平台还在努力实现第一个能正常运行的区块链应用，针对特定场景的区块链应用，诸如 BitShares 去中心化交易所（2014）和 Steem 社交媒体平台（2016）已经成为日活跃用户上万的成功应用。这两个应用成功地把性能提高到每秒可以处理数千个交易，延迟降低到 1.5 秒，交易费用也降低了，并实现了与中心化服务器方案相似的用户体验。

1. 千呼万唤始出来

2008 年爆发全球金融危机，当时有人用“中本聪”的化名发表了一篇论文，描述了比特币的模式。和法定货币相比，比特币没有一个集中的发行方，而是基于网络节点的计算生成，谁都有可能参与制造比特币，而且可以全世界流通，可以在任意一台接入互联网的电脑上买卖。不管身处何方，任何人都可以挖掘、购买、出售或收取比特币，并且在交易过程中外人无法辨认用户身份。比特币用分布式账本技术摆脱了第三方机构的制约，中本聪称之为“区块链”。区块链是一种共享的分布式数据库技术，具有四个显著的技术特点：

- 去中心化（Decentralized）：在一个分布有众多节点的系统中，每个节点都具有高度自治的特征。节点之间彼此可以自由连接，形成新的连接单元。任何一个节点都可能成为阶段性的中心，但不具备强制性中心的控制功能。节点与节点之间的影响，会通过网络而形成非线性因果关系。其没有中介机构，所有节点的权利和义务都相等，任一节点停止工作都会不影响系统的整体运作。
- 去信任（Trustless）：若任何一个区块链上的节点都无法造假，并且无法伪造，则两个陌生人的交易就不需要信任机制了，因为数据块上的信息随时可以被拿出来考证，这便是区块链的去信任机制。区块链能够完成价值转移和信用转

非卖品，仅供非商业用途或交流学习使用

移，从更加宏观的角度来看，这都是基于区块链是一种“去信任”架构实现的。“去信任”架构就是在整个系统中，多个参与方无须互相信任就能够完成各种类型的交易和协作，这恰恰一直是传统互联网到目前为止最薄弱的一项。因为数据库和整个系统的运作是公开透明的，在系统的规则和时间范围内，节点之间无法欺骗彼此。

- 集体维护（Collectively Maintain）：分布式整个系统中任意节点之间的权利和义务都是均等的，且任一节点的损坏或者失效都不会影响整个系统的运作。分布式具体指分布式记账、分布式传播和分布式存储这三大特性。系统中的数据块由系统中所有具有维护功能的节点来共同维护，而这些具有维护功能的节点是开源的，任何人都可以参与，这称为区块链的集体维护。
- 可靠数据库（Reliable Database）：系统中每一个节点都拥有最新的完整数据库拷贝，修改单个节点的数据库是无效的，因为系统会自动比较，认定最多次出现的相同数据记录为真。区块链实际上就是一个数据库，放在区块链里面的任何数据都是唯一且不可更改的。相对于银行较高的安全系数来说，区块链不需要额外的安全防御体系，不需要人力与物力就能够保证安全。

我们都知道比特币是一个纯粹的数字货币，它的作用仅仅是用来转账的，但是区块链技术本身的特点可以被用于更为广阔的空间与领域。目前最为成功的区块链平台是以太坊，但是以太坊的处理速度太慢，每秒只能处理 20～30 笔的交易。如果在以太坊的平台上开发一个类似淘宝或京东的应用，以这样的速度去处理交易，这对于一个商业级别的应用是无法忍受的。

2. 千锤万凿出深山

支持比特币的区块链技术并不能满足现实的需求，因为它 10 分钟才能产出一个块对其交易进行打包，这与现在银行的交易速度是无法比拟的。想要做到类似银行交易的速度，就需要改变所使用的共识机制，BM 决定引入一个新的共识机制——DPoS，以此来取代比特币所使用的交易速度较慢的共识机制——PoW。

2013 年，Bitshare 的第一版本上线，在 DPoS 共识算法的支持下大大提高了它的速度。由原来的 10 分钟产出一个块变为 10 秒产出一个块，尽管 Bitshare 采用了新的共识机制，但是却沿用了比特币的 UTXO 账户机制，导致 bug 层出不穷、要消耗大量系统资源及糟糕的用户体验，这让用户十分不满。

2015 年，为了更好地优化系统性能，BM 团队发布了石墨烯工具组，Bitshare2.0 采用了此技术之后，缩短了区块生产的速度，每 3 秒就可以产出一个块，同时支持每秒 1 万笔交易的处理速度，这也证明了分布式环境下的高性能，这些让 BTS 成为第一个去中心化的高性能交易平台。

2015 年 11 月，BM 离开 BTS。之后他将目光放在了内容社区上，并打造了一个通过加密货币奖励支持社区建设和社交互动的区块链数据库——Steem。Steem 沿用了比特股的底层技术：石墨烯区块链底层，在交易性能和用户体验方面都达到了相当高的水准。简单来说就是利用内容激励内容生产，作者可以根据点赞和转发量来获得 Token 奖励，这种“股东评价、系统奖励”的模式在当时大有挑战 Facebook、Twitter 的势头。BM 说：“ Steem 给我最大的激励是，有些在非洲的穷人因为我开发的软件而过上了中产的生活。这是让我每天早上起床的动力，当你有了一些钱之后，‘再去挣更多的钱’就已经不是什么重要的事情了。我会继续尽我最大的努力去找到一个自由市场的解决方案。”

2017 年，BM 离开了 Steem。作为一个社交媒体应用，Steem 也许是比较成功的，但作为其他功能就比较狭隘了。Bitshare 和 Steemit 都是基于区块链开发的应用，如果想向区块链添加新功能时，必须让所有参与者都达成共识，同时必须做大规模的升级，其间无尽的争议会耗时较多。BM 理想中的新系统应该有更好的治理架构以及权责安排。为此，他研究了博弈论，学了人们在类似投票的博弈过程中的各种纠结心理，最后发现这一系列东西都应该被重塑，可现在每个人都已经参与进这个系统，于是注定无法重塑它。所以，他决定去创建理想中的新系统。

EOS 可以说是 BM 对于区块链技术理解的总结，这次 BM 将以太坊作为自己的对手，建立一个区块链操作系统。一方面，经过多年的运行，石墨烯的技术可行性经受住了考验；另一方面，他有了更强大的团队，EOS 背后的 block.one 汇集了众多行业精英：CEO BB、合伙人 Iran 和 MichealChao。BM 只担任 CTO 一职，专心于开发工作，而不必再用他的极客思维去干扰项目运营。EOS 项目引入一种新的区块链架构设计，它使得去中心化的应用可以横向和纵向扩展。其构建了一个仿操作系统，在它之上可以构建应用程序。该软件提供账户、身份验证、数据库、异步通信和跨越数百个 CPU 内核或集群的应用程序调度等功能。由此产生的技术是一种区块链架构，它可以扩展至每秒处理百万级交易且用户可免费使用，很容易就能快速部署去中心化应用。

EOS 于 2018 年 6 月 2 号结束了长达一年的众筹，1.0 版本如期发布。6 月 15 号晚上主网投票率满足 15% 的启动条件，EOS 从此正式启动运行。EOS 此刻已经是一条独立运行的底层公链，或者是企业级区块链操作系统。就目前来说 EOS 要走的路还很长，但是终会有一天成长起来。请记住，EOS 创世块被匿名留的言——Never doubt that a small group ofthoughtful, committed citizens can change the world; indeed, it's the onlything that ever has.（不要怀疑，一小群深思熟虑、尽心尽责的公民能够改变世界。曾经如此，未来依旧。）

1.4　EOS 项目版本更迭简况与现状

从 EOS 项目在 github 上出现到本书截稿时为止，它已经存在 18 个版本了。可以看出为了它能更好发展，其团队在不间断地努力着，所以我们有理由相信 EOS 项目的前景会是更广阔的。接下来我们看看这些主要的版本更新了哪些功能。

2017 年 9 月 15 号，EOS 成功在 GitHub 上部署，这就意味着 EOS 作为一个开源项目可以与全世界共享。作为 EOS.IO Dawn 1.0 的第一个预发行版，它奠定了 EOS 的基本框架与基本的功能，以便于后期开发人员在此基础上做功能的修改与更新，Dawn 系列的 EOS.IO 软件版本代表了早期的 Alpha 质量软件，适合那些希望在 EOS.IO 生态系统上抢占先机的人们使用。在这个版本中主要涉及的功能有：

一个独立的节点 eosd，它产生块并将它们添加到区块链中。

2018 年 6 月 2 号，EOSIO V1.0 发布，这是 EOSIO 的第一个正式本版，之前的 Dawn 版本都是为正式版本做铺垫的，我们来看看作为第一个正式的版本又做了哪些功能的变动：

- 此版本引入了基于 boost::beast websockets 的替代 P2P 网络协议，该协议多线程网络和更快的同步。这个网络插件与现有的网络插件并排工作，但它只会与网络上的其他 bnet_plugins 通信。它完全独立且可选。要启动并运行新插件，只需在配置文件中或通过命令行包含 eosio::bnet_plugin。
- 黑名单 / 白名单，以确保链的核心功能（系统合同，令牌合同，多重合同和本机操作）都在一起工作。在打开链条之前，任何用户提交的合同都会大大增加攻击面。为了支持此功能，通过 nodeos 向链接插件添加了四个新选项。它们

是 --actor-whitelist、-action-blacklist、-contract-whitelist 和 --conract-blacklist。

- 还有一些小功能的更新，例如，在显示私钥时需要钱包密码；在 producer api 插件中公开运行选项；配置文件中不再支持 genesis-json 选项；增加了新配置选项 p2p-max-nodes-per-host；keosd 不再为不需要的操作启动；更改了端口 80 或 443 中的清除操作；将 max-irreversible-block-age 参数更改为无限制，通过 max-irreversible-block-age 参数设置区块链数据的 checkpoint，即不可逆区块高度；创建一个新账户需要购买 RAM 命令等。

2018 年 6 月 9 号，发布了 EOSIO V1.0.2 版本，这个版本的更新主要是为了提高系统可用性与安全性：

- 可用性：向 chain_plugin 添加 API 以获取任何可逆块的区块头状态；从 websocketpp 中获取有意义的错误消息；修复 15% 消息（系统合同中更改的错误消息）；当权限不足时可更好地处理错误消息。
- 安全性：已经正式使用 HackerOne 设置程序。

后面还有三个版本，这些都是针对可用性与安全性的更新，分别是 EOSIO V1.0.3、EOSIO V1.0.4、EOSIO V1.0.5，这里我就不详细说明了。

2018 年 6 月 23 号，发布了 EOSIO V1.0.6 版本，该版本主要解决主网上线后所暴露出来的一些问题，主要涉及的问题是：

- 在 nodeos 中：延迟事务（由具有相同的较新延迟事务隐式替换）sender_id 将不会释放以用于将替换的延迟事务存储回用户资源池的 RAM。
- 在系统合同中：用于存储延期退款交易的 RAM 账户被不正确地设置给接收方。

2018 年 6 月 27 号，发布了 EOSIO V1.0.7 版本，该版本解决了以下问题：

- 系统合约更新涉及：阻止无效账户名称出价；自我委托时不允许转移；修复可以从接收器中取出代币的错误。
- Nodeos 更新内容包括添加了对 msync 的额外调用，更好地保证了成功关闭 nodeos 的一致性；为 abi_serializer 解包数组添加额外的验证，允许 abi_serializer max_serialization_time 可配置，添加选项 --abi-serializer-max-time-ms；为功能添加了更严格的本地设置大小。
- Cleos 更新内容为 cleos set account permission 添加 R1 公钥选项。

2018 年 7 月 4 号，发布了 EOSIO V1.0.8 版本，该版提供了非关键的安全更新。

EOSIO 区块链的设计需要许多以特权级别运行的智能合约，以支持块生产者注册和投票、CPU 和网络带宽的获取、RAM 购买、Token 传输等多个功能。其他小的可用性改进有尝试创建名称已存在的账户时，更好地提示错误消息；尝试删除链接到操作的权限时更好地给出错误提示消息；引用不存在的权限时更正错误消息等。

2018 年 7 月 9 号，发布了 EOSIO V1.0.9，此版本解决了以下问题：HTTP 端点的主机标头验证。http-alias 参数可以重复指定，以此提供备用主机地址，http-validate-host 用来设置是否开启验证功能。用户在 cleos 更新后无法与 RPC 节点通信。

2018 年 7 月 13 号，发布了 EOSIO V1.0.10 版本，该版本解决了以下问题：

- 放松对主机标头验证。当通过 IP 连接时，如果 Host: 标头中的 IP 与套接字的本地 IP 匹配，Host: 则认为它是有效的。这允许正确操作 0.0.0.0 侦听地址；通过众所周知的端口字符串指定 PORT 时，该端口的解析版本在 Host: 标头中也有效。
- 适用 http-alias 于 Docker 服务。Docker 的典型用例是根据可解析的服务名称 nodeos:8000 访问它们。例如 Docker 配置现在可以正确地将这些服务名称添加到验证 HTTP 别名以及其他常见主机名 localhost 中。

2018 年 7 月 18 号，发布了 EOSIO V1.1.10 版本，该版本主要添加了以下功能：

- 添加了 MongoDB 作为一个可选性的插件，可以将区块链数据存档到 MongoDB 中。这将在区块链中创建数据的只读副本，从而实现对数据的高度可扩展，从而使数据方便查询。
- 通过代码优化使 nodeos 更快同步账本数据。当节点因距离太远等因素导致同步失败后，将不再尝试中继事务。

2018 年 7 月 25 号，发布了 EOSIO V1.1.1 版本，该版本提供了先前向现有基于 EOSIO 的区块链保管人以及非关键安全更新披露的关键安全更新。

- 取消延期交易：使用“retired”WRT 等语义来执行事务。
- 添加 Net 插件以改进对传入区块请求的健全性检查。

2018 年 8 月 2 号，发布了 EOSIO V1.1.2 版本，该版本解决了以下问题：

- 错误修复和次要改进：将 LICENSE.txt 重命名为 LICENSE ；隐式安装到系统范围的路径时添加警告与更正；在构建期间更好地发现核心 CPU 数量；删除重复的块签名检查；改进二进制文件和合同开发依赖项的安装。

- 非关键安全更新：优化 bnet_plugin 中的 block_status 跟踪；优化 net_plugin 中的连接释放；将未对齐的数组副本移动到堆；以 cleos 方式序列化事务而不是使用 RPC。

以上是对 EOS 以往版本进行的简单描述，因为就目前来讲，EOS 已经有 64 个版本，在这里我就不一一列举了。这里只是简单介绍了一下 EOSIO 正式上线之后的版本情况。当然，就目前 EOS 现状来说还有很多功能需要进行完善，比如其每秒可处理的交易量并没有达到宣传的数值、在之后可能会出现分叉等问题。这个科技快速发展的今天，对 EOS 来说既是机遇又是挑战。

第 2 章 Chapter 2

EOS 项目技术优势解析

EOS 是 EOS 软件引入的一种新的区块链架构，旨在实现分布式应用的性能扩展，相比于目前区块链普遍存在的平台性能低、开发难度大及手续费高等问题，EOS 拥有高性能处理能力，以及易于开发、用户免费等优势，可以极大满足企业级的应用需求，被誉为继比特币、以太坊之后的区块链 3.0 技术。

EOS 的主要特点如下：

- EOS 类似于 Windows、Linux 平台，通过创建一个对开发者友好的区块链底层高性能平台，支持多个应用同时运行，为开发 DApp 提供底层模板。
- EOS 通过并行处理和 BFT-DPoS 共识机制解决了延迟和数据吞吐量的难题，EOS 目标是每秒可以完成百万级别的交易处理量，而比特币每秒仅可处理 7 笔交易左右，以太坊每秒仅可处理 30～40 笔交易；
- EOS 没有手续费，普通受众群体更广泛。在 EOS 上开发 DApp，需要用到的网络和计算资源是按照开发者抵押的 EOS 的比例分配的。当用户拥有了 EOS Token，就可以在 EOS 系统换算计算机资源。随着 DApp 的开发，用户可以将手里的 EOS Token 租赁给别人使用，单从这一点来说 EOS 也具有广泛的价值。简单来说，就是用户拥有了 EOS，就相当于拥有了一套房以租给别人收房租，或者说拥有了一块地以租给别人建房。

仅供非商业用途或交流学习使用

本章主要对 EOS 项目技术做详细解析，并分析其技术上的优势。

2.1 EOS 的共识机制（BFT-DPoS）

BTC 和 ETH 采用的共识机制都是 PoW（Proof of Work，工作量证明机制），由于该机制需要矿工消耗电力资源挖矿，而且其共识算法效率较低，很难进一步满足全球范围内的交易量和上层应用开发的大交易量需求。

EOS 项目早期时候使用的共识机制是 DPoS（Deligated Proof of Stake，委托股权证明），类似于 Bitshares 和 Steem 采用的共识机制，该共识机制类似于西方国家的议会制度，共识效率比较高。

DPoS 主要由五大部分组成：Token、区块链、社区、计算机和规则。在整个区块链系统中，Token 代币的持有者以 Token 为选票，通过投票选出其认可的节点，选出的节点根据一定的规则来运行区块链计算机网络。在 EOS 网络中，最终会选出 21 个主节点，它们有权力生产区块、这些主节点被称为区块生产者，简称 BP（Block Producer）。

DPoS 共识机制采用随机的 BP 出块顺序，出块速度为 3 秒 / 块，每一个 BP 的出块都代表了对之前整条区块链进行了一次信息确认。由于交易需要得到全网 2/3 以上 BP 的确认，在 EOS 中即需要得到 14 个 BP 的确认，因此确认交易的不可逆总共需要 45 秒（15*3），如图 2-1 所示。

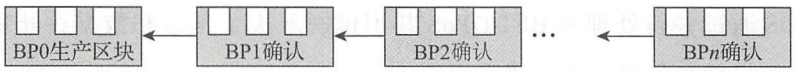


图 2-1 DPoS 交易确认示例图

在传统的 DPoS 中，BP 在出块时在全网广播该区块，其他 BP 即使收到了目前的新区块，也无法立即进行确认回复，而是需等到轮到自己生产区块时才能进行确认。由于该方法交易的确认时间过长（确认交易至少要 45 秒），为了提升整个网络的性能，BM 借鉴了 Practical Byzantine Fault Tolerance（拜占庭容错算法）的机制，采用了 BFT-DPoS 共识机制。

在新共识机制下，BP 生产新的区块时依然向全网络广播，其他 BP 收到新区块时，

仅供非商业用途或交流学习使用

立即对此区块进行验证，并将验证完成的签名区块立即返回给出块 BP，不需要等到轮到自己出块时再确认。因此在生产新的区块并全网广播之后，在收到全网 2/3 的 BP 确认后，该区块就不可逆了。该机制下的交易确认时间由原先的 45 秒缩短至 3 秒，消耗时间主要为等待生产区块的时间。该共识机制过程如图 2-2 所示。

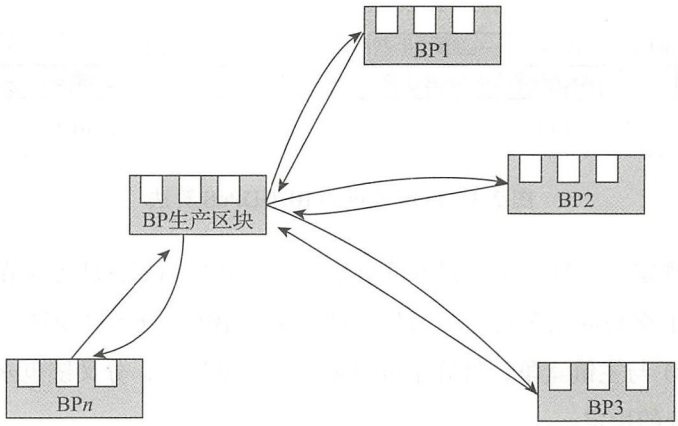


图 2-2　BFT-DPoS 交易确认示例图

为了继续挖掘 EOS 的性能，BM 又对以上机制进行了修改。他将出块速度由原来的 3 秒缩短为 0.5 秒，这样可以极大地提升系统性能，但该做法也会带来系统的网络延迟问题：由于网络延迟的存在，0.5 秒的确认时间可能会导致下一个出块的节点还没有收到上一个出块节点的区块即产生新的节点，从而导致区块链分叉。例如：中国一节点生产区块后紧接着轮到美国一节点，但因中美网络的延迟高达 300 毫秒，在美国的节点要出块时还没收到中国节点的区块，则会忽略中国节点产生的区块，形成区块链分叉。

针对上述问题的不可避免性，BM 将原先的随机出块顺序改为由 BP 商议后确定的出块顺序，这样网络延迟较低的 BP 之间相邻出块，从而可尽量避免区块链分叉。

同时为了更有效地解决上面的问题，不让任何一个 BP 因为网络延迟而被跳过生产，BM 设计让每个 BP 生产区块时连续生产 12 个区块，即拥有 6 秒的连续出块时间。这样的机制能够保证即使在最恶劣的情况下，BP 连续生产的 12 个区块中，只有最后一个或者两个区块可能因为网络延迟而被下一个 BP 忽略，但能保证前列的区块能够有效传递给下一个 BP。

仅供非商业用途或交流学习使用

现有的完整 BFT-DPoS 机制如下：每个区块拥有 6 秒的出块时间（能够连续生产 12 个区块），区块生产后立即对全网进行广播，区块生产者一边生产新的区块一边接收其他 BP 对于上一个区块的确认。在大部分情况下，交易会在一秒以内确认（即达到不可逆状态），该时间包含了 0.5 秒的区块生产和 0.5 秒的其他 BP 的确认，如图 2-3 所示。

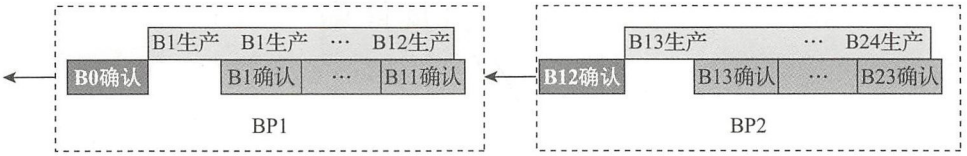


图 2-3 EOS 改进的 BFT-DPoS 算法

EOS 系统规定，一旦区块达到不可逆状态（2/3BP 确认），就无法在此之前进行分叉，从而保证了交易的永久可信。另外，即使多数 BP 想分叉区块链，也只能以相同的速度（0.5 秒）与主链竞争，就算主链只剩下一个见证人，分叉链也永远不会追上主链，保证了系统的稳定。

2.1.1 超级节点竞选

所谓超级节点，就是指 EOS 网络中那些收集交易信息并打包到区块里的节点，又称区块生产者（Block Producer，BP）。超级节点主要负责以下事务：

- ❑ 收集网络里的交易；
- ❑ 验证交易并把交易打包到区块；
- ❑ 广播区块给其他节点，通过验证后将区块添加到自己本地区块链上。

与支撑 BTC 和 ETH 的 PoW（工作量证明共识机制）不同，EOS 网络采取了 BFT-DPoS 的解决方案，该方案能够很好地解决前者延迟和低吞吐量的痛点。但另一面，EOS 也为自己带来了隐患——要想提升整个网络的效率，就必须减少节点，然而过少的节点又会对整个网络的安全性造成影响。

因此，EOS 设计者 BM 提出了“超级节点”的解决方案。超级节点一共有 21 个，为整个网络提供计算和带宽支持，每一个超级节点都组织了自身的 EOS 社区，而超级节点也会像现实中的城市一样去竞争发展。该方法能够以有限的中心化实现广义的去中心化，并让 EOS 获得极快的交易速度与容错能力。

仅供非商业用途或交流学习使用

EOS 超级节点的产生是根据全网用户持有的 EOS Token 投票实现的。不同于以太坊的 POA 一个节点一票的形式，EOS 是通过持币数量来决定手里的票数。因为以太坊的 POA 是基于小场景的私链或者联盟链的，这与大规模对外公开的公链 EOS 的场景是不同的。

节点被选举成功，以太坊 POA 是没有确定数量限制的，随时按照全网投票与票数清零以后的每一轮投票结果去增删超级节点。而 EOS 超级节点只有 21 个，在整个投票周期结束以后，排名前 21 的节点即胜任。并且这 21 个超级节点必须符合非常高的性能要求、运维能力和社区规模等，当某个超级节点出块方面出现问题时，在一定规则下其会被丢弃，然后重新选出新的节点替代它作为超级节点的身份。

2.1.2　持币者投票

主网上线后，EOS Token 持有者可以在已支持 EOS 的钱包中或者其他插件中进行 EOS 节点的投票。其投票流程如下：

1.EOS Token“抵押”

持有者要获得选票，首先需要将手里的 EOS Token 抵押，转换成相应的 CPU 和带宽资源（NET），可投的选票数与抵押的可用资源（CPU+NET）相等。这里需要注意的是，EOS 系统中通过抵押或者购买得到的 RAM，只能获得 RAM 的使用权而不具有投票权。抵押的 EOS 会被锁定，不再具有转账和交易等其他功能，如果想取消抵押并赎回 EOS Token，则在系统中经过赎回操作后需要再等待 72 小时，EOS Token 才能回到自己账户上，即赎回抵押的 EOS Token 具有 72 小时的锁定期。

2. 一票 30 投、委托代理

在 EOS 系统中，通过上述抵押获得的选票最多可以投给 30 个不同的节点，假如你持有 1 个 EOS Token 对应的选票，那可以将这一票投给一个节点，也可以投给 30 个节点，这 30 个节点都会获得 1 选票。如果想修改自己的投票，可以立即进行改投，更改会在下一轮的区块开始生产时生效（每一轮区块生产时间为 21×6 秒）。

还有一种投票方式为委托代理，即你可以把投票的权利交给指定的投票代理人，让他来代替你进行投票。委托代理机制对那些想要投票，但无法了解全部节点候选人的用户来说是非常有用的。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

3. 投票权重衰减

系统为了推进整个系统的治理更新，希望持有 EOS Token 的用户能够根据节点的发展运营情况经常更新他们的投票，为此引入了一个时长为一年的投票权重半衰期机制。该机制规定从投票一周后开始，如果用户在一周后不重新投票，该选票的权重就会衰减，一年后的权重会降低至初始的 50%。如果用户重新投票，其选票的权重就会完全恢复。

在每一轮区块生产时，系统会根据每个节点的选票数和每张选票的权重值得到各个节点的选票值，根据排名筛选出前 21 个节点作为系统的超级节点。

2.1.3 超级节点治理

超级节点可以类比理解为 PoW 系统中的矿工。EOS 中区块产生的过程：EOS 的区块生产过程和比特币系统类似，都是收集网络中的交易信息并将其打包为区块。不同的地方是，EOS 中的超级节点不需要完成工作量证明即可直接广播区块到网络中，让其他节点进行验证。

在 EOS 系统中，每生产 252 个区块为一个区块周期，即每个超级节点在一个区块周期中应该产生 12 个区块。每个区块周期内，EOS 系统设置了每隔 0.5 秒产生一个区块。在每个区块周期开始时，EOS 会根据代币投票结果选出 21 个超级节点。这 21 个超级节点会约定一个区块产生顺序，这些节点按照这个顺序每隔 0.5 秒产生一个区块。如果一个节点在指定的时间内没有产生出一个区块，那么这个区块就会被跳过，并在区块链上留下时间戳间隔。如果一个节点在最近的 24 个小时内都没有产生区块，那么它将被移出超级节点的候选名单。

一旦一个节点完成了区块生产并通过其他节点的验证，它就可以将这个区块添加到区块链上，并获得新的代币作为出块奖励。白皮书中规定，所有超级节点的期望奖励的中位数决定了系统最终会产生多少新代币。EOS 系统有一个硬性的条件，即每年产生的新代币数量不能超过代币总量的 5%。BM 在 DAWN-446 提议中认为，应该将生产奖励分为三部分，分别作为超级节点区块生产奖励、备用节点奖励和新提案奖励。

2.2 EOS 的账户体系

账户体系是 EOS 中的亮点之一，实现了基于角色的权限管理和账户恢复功能，使

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

得用户可以灵活地以一种组织化的方式管理账户，并最大化保证资产的安全性。

2.2.1　基于角色的多层级账户体系

常见区块链项目的账户是一对公私钥，账户名就是公钥，而 EOS 中的账户名是由用户自定义的 12 位可读标识符组成，且一般一个账户下面包含多对公私钥，每对可以自定义拥有不同的权限。通过权限配置可实现该账户只被个体拥有或者被一个组织控制（即多个个体共同拥有），该账户是传统公钥所代表的单一权限的更高层次抽象集合。

对于常见的区块链项目而言，一般由于只有一对公私钥，因此项目的代币会直接放在公钥里，如果私钥被盗取代币便可直接转走。而在 EOS 中，代币是放在账户里的，公钥里面放的是带权重的钥匙，转走账户里面的代币需要每个拥有转账权限的钥匙的权重之和达到设定的阈值才可进行。EOS 上的所有交易行为都是通过账户来完成的，通过账户执行任意操作时，EOS 系统首先会验证操作者是否拥有足够的权限，验证通过该操作才能生效。

EOS 中，每个账户刚创建时一般由个体拥有，通过单一公私钥便能进行所有操作，后续可根据需要通过权限配置将该账户扩展成组织账户，由多对公私钥（即多个主体）共同控制，甚至可为组织外部个体或组织分配部分操作权限，从而实现极其灵活的组织管理方式。

2.2.2　账户权限管理

EOS 中任意账户都自带两个原生权限：owner 和 active 权限。owner 即代表账户所有权，该权限可进行所有操作，包括更改 owner 权限，可由一对或多对 EOS 公私钥或另一账户的某权限实现权限控制。因此，代表着 owner 权限的 EOS 公私钥是最重要的，必须冷储藏。active 即活跃权限，能进行除更改 owner 权限以外的所有操作，也是通过一对或多对 EOS 公私钥或另一账户的某权限实现权限控制的。

EOS 支持将 A 账户的某一操作权限分配给他人所拥有的公私钥对或者账户 B，从而实现基于角色的权限管理。

除了两个原生权限以外，EOS 还支持自定义权限。active 权限可以看作拥有除了更改 owner 权限外的所有 owner 权限，例如转账、投票、购买 ram 等权限。我们可以将 active 的部分权限（比如说投票权）分配给一个自定义权限 voting，且无须 owner、

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

active 权限所对应的私钥对投票操作进行签名，单单通过 voting 权限所对应的私钥对投票操作进行签名便可完成投票操作。这就可以将 EOS 账户的部分操作权限分配给第三方，避免直接给出 active 权限的私钥，从而实现灵活和安全的组织管理方式。

在多主体共同控制某一权限的情况下（如多对 EOS 公私钥共同控制 owner 权限），如何判定，或者说在何种条件下就拥有了该账户的某一权限？ EOS 是通过权重和阈值来实现的。账户可给每个主体（如每对 EOS 公私钥）分配不同的权重，以及拥有该权限的阈值，只有当某些人拥有的公私钥数量所对应的权重之和不低于该权限的阈值时，才能拥有该权限，并进行相应操作。下面以某账户的权限配置为例进行详细说明，如表 2-1 所示。

表 2-1 账户权限配置

权 限	阈 值	控制该权限的公钥或账户	权重
Owner	2	EOS2Ca4o...	1
		EOS3Q3bx...	1
active	2	EOS94x3B...	2
		EOS4x112...	2
voting	2	Testaccount1@active	1
		Testaccount2@active	1
		EOS7Hn1p...	2

上表中，该账户的 owner 权限由 EOS2Ca4o... 和 EOS3Q3bx... 公钥所对应的私钥的所有拥有人共同控制，任何一方都不能单独拥有 owner 权限。因为双方的权重都为 1，而 owner 权限的阈值为 2，因此只有双方达成一致，一起对交易进行签名才能行使 owner 权限。

而 active 权限则可由 EOS94x3b... 和 EOS4x112... 任意一方单独行使。

对于自定义权限 voting，由 EOS7Hn1p... 公钥所对应的私钥拥有的人单独控制，而账户 testaccount1 则需和账户 testaccount2 通过各自的 active 权限结合起来才能共同行使该账户的 voting 权限。

2.2.3 强制延迟消息执行策略

时间是安全的关键组成部分。在大多数情况下，不可能知道私钥是否被盗用，直

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

到它被使用。基于时间的安全机制在人们使用某些应用程序时更为重要，因为这些应用程序需要将密钥保存在日常使用的联网计算机上。消息包含在区块后，EOS 系统软件支持应用程序开发人员指定某些消息在应用前必须等待一小段时间，在此期间可以取消该操作。当这类消息被广播时，用户可以通过电子邮件或短信收到相应通知。如果用户没有授权，那么他们可以登录账户来还原账户数据并撤回消息。所需的延迟取时间决于操作的敏感程度。支付一杯咖啡可以在几秒钟内确认且不可撤回，而买房子可能需要 72 小时的清算周期。将整个账户转移到新的用户可能需要 30 天。具体延迟取决于应用程序开发人员和用户。

该机制的存在可以使开发人员结合具体场景有效保证用户的资产安全。

2.2.4　密钥丢失或被盗后的恢复

EOS 允许恢复被盗窃的密钥，这在比特币和以太坊上是不可能的。在比特币和以太坊上一旦密钥丢失那么整个账户也将随之丢失。EOS 系统软件为用户提供了密钥被盗时恢复其账户控制的方法。账户所有者可以使用过去 30 天内活跃的任何其批准的账户恢复合作伙伴的密钥，并重置账户上的所有者密钥。没有账户所有者的配合，账户恢复合作伙伴无法重置账户的控制权。

对于黑客而言，由于其已经“控制”该账户，因此尝试执行恢复过程没有任何收获。此外，如果黑客执行恢复过程，指定的恢复伙伴可能需要身份认证和多因素认证（电话和电子邮件）。这可能会暴露黑客身份，或者黑客在恢复过程中毫无所得。这个过程也与简单的多重签名交易有着极大的不同。通过多签名交易，另一个实体会成为每个执行交易的一方。相比之下，通过恢复过程，恢复合作伙伴仅参与恢复过程，无权参与日常交易。这极大降低了所有参与者要付出的成本和相应法律责任。

2.3　网络资源 Token 化

2.3.1　带宽和日志存储

用户发送一笔交易信息后，区块生产者需要将交易打包生成区块，然后将区块通过网络同步给其他生产者，这个过程需要消耗一定网络带宽资源。而带宽资源由交易信息在区块中所占的字节数决定。带宽资源的计费方式类似手机流量，用户每发送一

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

笔交易信息就消耗一点带宽资源，如果带宽资源消耗为 0，则无法继续发送交易信息。

获取带宽资源的方式有两种：

（1）如果用户账户持有 Token，那么可以直接将 Token 抵押给 EOS 系统账户，系统会根据抵押 Token 所占全网 Token 的比例，分配给用户对应的带宽资源。比如，用户持有 1% 的 Token，那么该用户就拥有全网 1% 的带宽资源。同时，用户可以取消抵押，将带宽资源转换为等量的 Token。这种带宽获取方式本身不会消耗 Token，但是，已经抵押的 Token 不能够继续抵押或者出售，抵押锁定期至少为 3 天。

（2）如果用户账户没有 Token，可以从其他用户手中租赁带宽资源。

2.3.2 计算和计算积压（CPU）

当用户调用智能合约时，区块生产者需要根据智能合约地址查找合约代码，然后将代码加载到内存中执行，这个过程需要消耗一定的 CPU 算力。CPU 计算资源由运行智能合约（交易也属于智能合约的一种）所消耗的时间决定，用户每次调用智能合约都会消耗一点计算资源，如果消耗为 0 则无法继续执行合约。计算资源也是通过抵押 Token 或者向其他用户购买获得。解除抵押之后，Token 也需要至少锁定 3 天。

2.3.3 状态存储器（RAM）

RAM 是指运行时的内存，在 EOS 系统中，用于存储账户等状态信息，是 DApp 运行的基础资源。根据账户所抵押的 EOS 数量的不同，账户能够调用的资源也是有所不同的。跟 Steem 一样，EOS 使用了速率限制的算法来计算账户可以动用的资源。

RAM 的买卖，实质上是抵押 EOS Token 到系统账户，就是说将 EOS Token 转账到 eosio 这个系统账户之中作为抵押，而不是买方和卖方直接的交易。购买 RAM，是抵押 EOS 得到了 RAM 的使用权限；而出售 RAM，则是释放了自己所占用的 RAM，从系统账户之中取回所抵押的 EOS。和网络带宽资源不同的是，RAM 涉及的抵押或者解除抵押需要收取 0.5% 的手续费，并且获得 RAM 不能出租给其他人，只能给自己使用，抵押或取回 RAM 是立即进行的，不像 CPU 和带宽资源一样，有 3 天的延迟期。RAM 的价格采用 bancor 算法，其核心思想就是根据市场的供求关系决定 RAM 价格，买的人越多，价格越高。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

RAM 的总量由超级节点共同投票决定，一旦确定之后，短时间内不会再发生大幅扩容情况。而 RAM 的需求端由 DApp 市场需求决定，当 DApp 上线运行后，会一直锁定内存，不会释放。目前市场的 RAM 价格昂贵，所以，RAM 会成为一种稀缺资源。相比之下，带宽资源和 CPU 计算资源会随着时间推移，由超级节点源源不断地提供，所以不具备明显的稀缺性。

2.4　EOS 的节点治理

EOS 是为商用分布式应用设计的一款区块链操作系统，以后会有很多在 EOS 上开发的 DApp，不同的应用难免会有许多不可预见的问题，而这些问题不可能完全在系统软件层面被解决。所以，在遇到这方面问题的时候，EOS 提前做好了解决方案——EOS 的节点治理。而目前存在的区块链项目没有这部分解决方案，例如我们熟悉的比特币和以太坊，因为没有这样的解决方案，社区经常因为一些原因分裂，进而导致分叉。2017 年比特币已经被分叉无数次了，以太坊也已经出现了数次分叉，而这样的结果就是没有考虑到那些在软件层面之外的问题所造成的。

2.4.1　宪法和协议指导下的关键操作

EOS 操作系统使用区块链技术在签名用户之间建立 P2P 服务协议或约束性合约，也就是所谓的“宪法”。宪法内容定义了仅依靠代码无法完全执行的用户间义务，同时结合相互间的公认规则，确立司法权和适用的法律。每一个在网络中签名广播的交易，其签名信息中必须包含宪法的哈希值，以明确约束合约签名者。

EOS 系统中，真正治理的权力在 Token 持有者手中而不在于区块生产者 BP 手中。Token 持有者可以将自己的权利代理给区块生成者，这样区块的生成者就有了相应的权限（冻结账户、更新有缺陷的应用程序、提出对底层协议硬分叉的改变等）。权限代理出去后，区块生成者并不是就可以随便乱用权限，那些权限是受限的、被监督的、被检查的。所有的区块链变更都需要区块生产者同意，如果区块生产者拒绝 Token 持有人想要的变更，那么他将被投票出局；如果区块生产者的变更没有经过 Token 持有人的同意，那么其他非区块生产者的全节点会拒绝该改变。

非卖品，仅供非商业用途或交流学习使用

- 冻结账户：一个智能合约可能会出现异常，比如说因为 bug 的原因导致行为不正确或者资源消耗不在一个合理的范围内，这时候区块生产者就有权力冻结账户。冻结后这个账户所做的行为就没用了。当然也不是随便就能冻结，冻结账户需要 15/21 的区块生产者同意才行。如果区块生产者滥用权力，解决方案也很简单，就是将他投票出局，这样被冻结的账户就会被解冻。
- 改变账户代码：如果“冻结账户”已经不能解决问题，不可预知的代码（病毒）已经造成了破坏，此时 EOS 可以支持在不需要硬分叉的前提下修改账户代码。这类似于交易的回滚。当然与冻结账户类似，也需要 21 个中的 15 个区块生产者同意才行。

2.4.2 宪法和协议的升级

“宪法”会随着当地环境的变化而变化，所以 EOS 也提供了修改的方法。EOS 操作系统使用源代码定义宪法和协议，同时也定义了宪法及协议的更新方法。对宪法或协议进行变更，需要完成以下步骤：

（1）区块生产者 BP 提交一个宪法变更协议，并获得 15/21 以上的赞成票。

（2）区块生产者 BP 将 15/21 以上的赞成票维持连续 30 天。

（3）要求所有用户都使用新宪法的哈希值确认交易。

（4）区块生产者 BP 采用修改源代码的方式反映宪法的变更，使用 git 提交的哈希值将变更提交到区块链上。

（5）区块生产者 BP 继续将 15/21 以上的赞成票维持连续 30 天。

（6）变更的代码 7 天后生效，源代码修改通过后，将有 1 周的时间来对所有节点进行升级。

（7）所有没有升级为新代码的节点将自动关闭。

根据 EOS 操作系统的默认配置，更新区块链来添加新功能这一进程需要 2 到 3 个月的时间，而修复那些不需要更改宪法的非关键性漏洞需要 1 到 2 个月的时间。

一般情况下变更所花费的时间比较长，当遇到特殊情况的时候，EOS 也提供了解决方案。紧急变更——面临一个损害用户利益的有害漏洞或安全漏洞时，区块生产者可以加速宪法变更过程。一般来说，加速新特性更新过程或修复无害漏洞，都是违反宪法的行为。

非卖品，仅供非商业用途或交流学习使用

2.5　脚本与虚拟机

1. 模式定义的消息

在账户之间发送的所有消息都是由区块链共识状态的架构来定义的，该架构允许消息在二进制和 JSON 格式之间无缝转换。账户和账户之间发送消息使用的是二进制的方式，这样更具高效性和易操作性。二进制文件对于人类来说极难看懂，所以 EOS 支持将数据转化为人类可读的 JSON 格式的字符串。

2. 模式定义的数据库

数据库状态也可使用架构定义，这可确保所有应用程序存储的数据都以一种格式呈现，同时具备 JSON 的人类可读性，以及二进制格式的高效率存储和易操作性。和模式定义的消息类似，模式定义的数据库使用二进制存储数据，但是支持转换成 JSON 格式，供人类理解。

3. 身份验证与应用程序分离

EOS 通过并行执行机制来实现高性能。为了实现高性能，EOS 一直都在考虑怎样进行并行和串行的区分。为了最大化并行运算，将从程序日志中重新生成应用程序状态的计算任务降至最低，EOS 操作系统将验证逻辑分为三个部分：

- 验证消息在内部是一致的：是只读的，不需要访问区块链状态，以最大的并行度来执行；
- 验证所有的前置条件都是有效的：是只读的，同样并行执行；
- 修改应用程序状态：需要写权限，因此必须顺序执行。

验证消息内部的一致性是只读的，不需要访问区块链状态，这意味着它可以通过最大化并行运算来执行。验证前置条件（例如需求平衡）也是只读的，因此也可以从并行运算中获益。只有对应用程序状态进行修改才需要写访问，并且需要按顺序对每个应用程序进行处理。

消息认证是一个验证消息可被使用的只读过程，而修改应用程序状态是写操作，EOS 将它们分开了。一旦消息被包含进区块就不再需要进行消息验证的操作了。

4. 虚拟机独立架构

EOS 操作系统的目的是可以支持多种虚拟机，同时可以随着时间推移持续按需求

增加新的虚拟机。理论上只要符合性能、确定性、正确性和沙箱化这几个条件的任何虚拟机都可以跟 EOS 进行对接。

目前公布支持的虚拟机有三个：

- Wren：Wren（http://wren.io）是一种小型、快速、基于类别的编程语言。Wren 的开发人员将其描述为“就像是把小型 TAK 文件装进 Lua 大小的软件包，再加上一点 Erlang 特性，再包进一个熟悉的、现代的语法里面”。之所以选择 Wren 语言和虚拟机，是因为它短小精悍、易于文档记录和理解的代码库。它还具有非常好的性能，并且可以很容易嵌入 C++ 应用程序中。
- Web Assembly【 web 组件 】（WASM）：WASM 是构建高性能 Web 应用程序的新兴 Web 标准，通过少量适配就可以被明确定义和沙箱化。WASM 的好处在于被业界广泛支持，因此可以用熟悉的语言开发智能合约，例如 C 或 C++。以太坊已经开始适配 WASM，以提供适当的沙箱并使用以太坊 WASM 定义（https://github.com/ewasm/design）。这种方法很容易改编后用于 EOS 系统软件集成。
- 以太虚拟机（EVM）：这个虚拟机已经被用于大多数现有的智能合约，并且可以在 EOS 系统区块链上使用。可以想象，在 EOS 操作系统的区块链上，EVM 合约可以在内部沙箱中运行，只需要进行少量适配就可以与其他 EOS 应用程序交互。

第 3 章 Chapter 3

EOS 代码原理分析

3.1 EOS 项目代码框架总览

我们要对 EOS 项目的底层代码有更深层的认识，就需要对 EOS 的整个框架进行了解，知道对应代码的作用。只有熟悉各个模块的作用，才可以把自己的应用灵活部署在 EOS 项目上，进而实现自己想要的效果。本章将 EOS 项目分 4 个层次来详细介绍。

3.1.1 脚本层

脚本层可为支持层和插件层提供基础能力，实现区块链的底层关键技术，例如交易处理、生产区块、加密功能、文件 IO 操作、网络通信等能力。EOS 项目的整体编译、依赖库下载、代码编译等功能都依托于一个完整的脚本体系。脚本层主要包括 eos/eosio_build.sh、eos/scripts、eos/CMakeModules 三部分，如图 3-1 所示。

1.eos/eosio_build.sh

当我们要把 EOS 项目完整下载到本地时，首先需要执行 eosio_build.sh 脚本文件。eosio_build.sh 是 EOS 项目的主编译脚本，通过运行这个脚本，就可以实现对 EOS 项目的一键编译，脚本运行过程中主要测试了编译所需的硬件与软件环境是否满足要求以及显示编译结果。

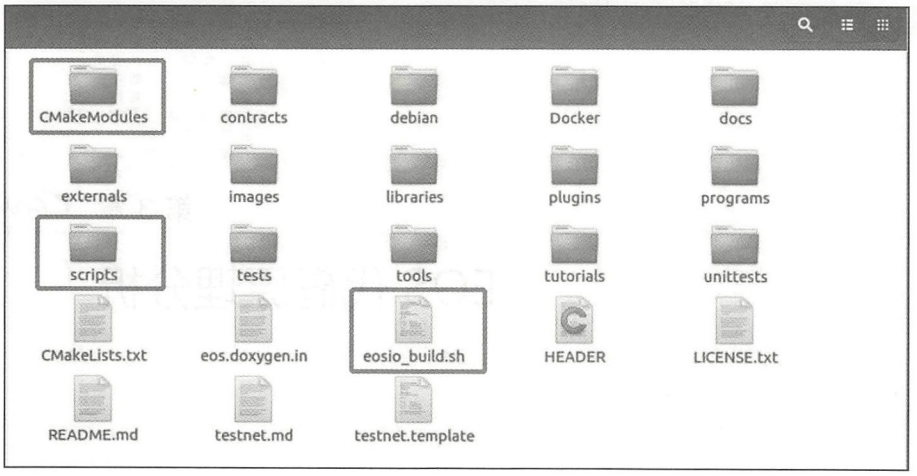


图 3-1 EOS 项目完整的预览图

2.eos/scripts

看到 scripts 这个目录的名字就会知道里面包含了项目编译所需的其他脚本文件，它们的作用分别是：abi_to_rc、abigen.sh 和 abi_is_json.py 共同负责将 C++ 语言编写的智能合约编译成 .abi 文件，再将 .abi 文件编译成可执行文件；eosio_build_ 前缀的脚本分别针对不同系统的编译子脚本；eosio-tn_ 前缀的脚本负责自动化运行，关闭节点。具体如图 3-2 所示。

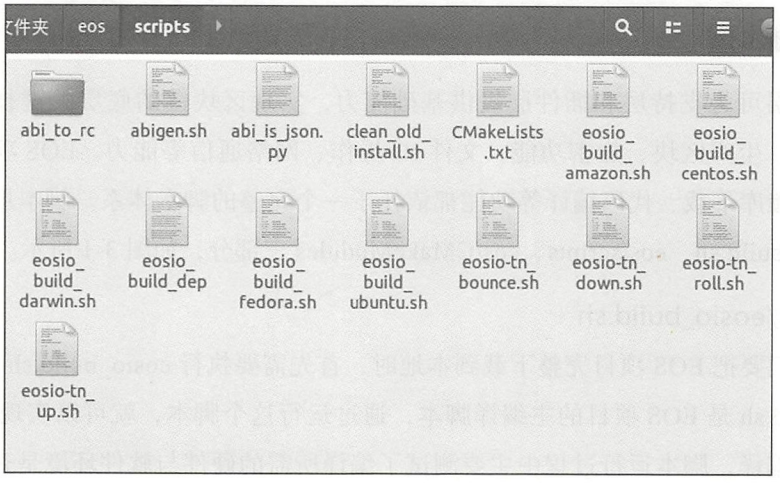


图 3-2 scripts 文件夹

3.eos/CMakeModules

由于 EOS 项目是基于 CMake 等工具编译的，所以 eos/CMakeModules 中主要包含 CMake 编译需要使用的一些配置信息。

3.1.2　插件层

EOS 项目中的节点程序依赖一个完整的插件体系，通过不同的插件组合来提供各种服务功能，如区块链查询、交易验证执行、打包区块、P2P 网络通信等服务。插件完整的生命周期包括注册、初始化、开启、停止四个流程。EOS 项目的插件代码位于 eos/plugins 目录下，包括模板层、基类层、派生类层、封装层四个层次，如图 3-3 所示。

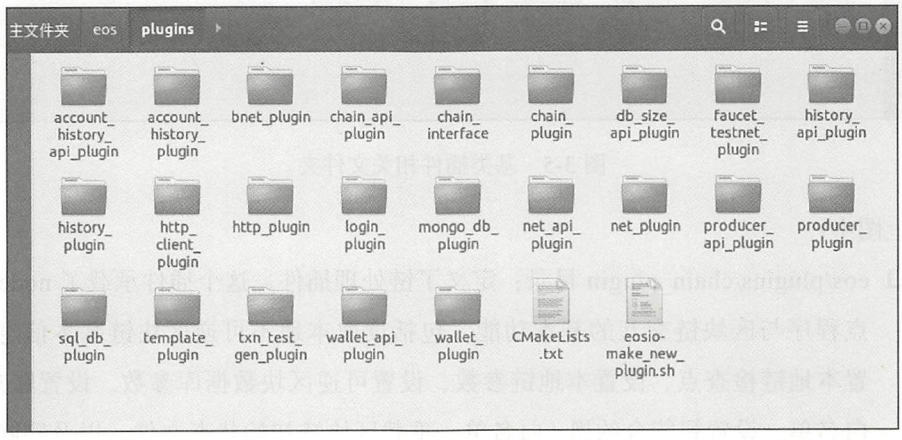


图 3-3　plugins 文件夹

1. 插件模板

eos/plugins/template_plugin 定义了 EOS 项目中所有插件的模板。图 3-4 所示为插件模板在 EOS 项目中的位置，其中 template_plugin.cpp 为插件模板源码，CMakeLists.txt 为编译代码。

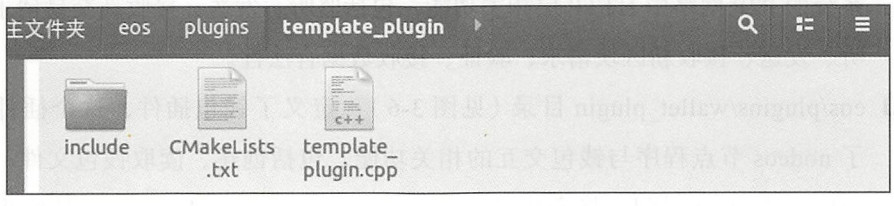


图 3-4　template_plugin 文件夹

2. 基类插件

EOS 项目的插件体系中共有 5 个基类插件，如图 3-5 所示。

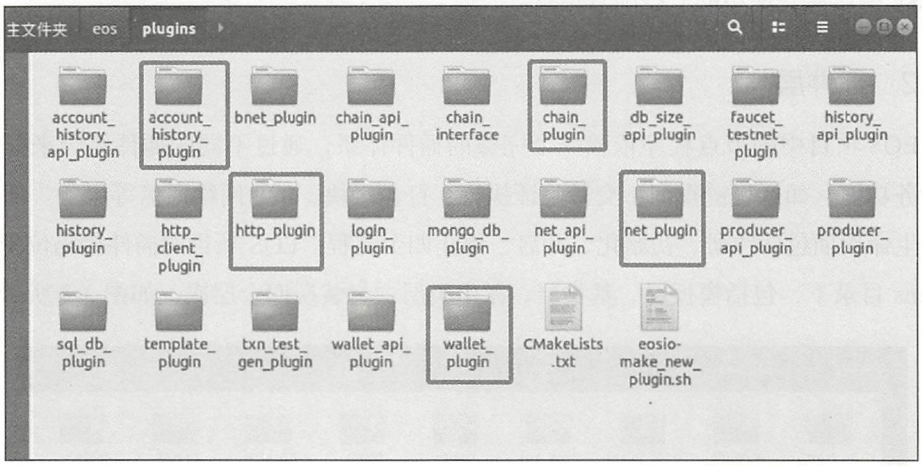


图 3-5 基类插件相关文件夹

上图中：

- eos/plugins/chain_plugin 目录：定义了链处理插件，这个插件承载了 nodeos 节点程序与区块链交互的基本功能，包括读取本地不可逆区块链基本信息、设置本地链检查点、设置本地链参数、设置可逆区块数据库参数、设置账户黑 / 白名单、设置智能合约黑 / 白名单、重载区块链初始状态文件，以及删除、重写、替换本地区块链数据（包括开始、停止等动作）。
- eos/plugins/net_plugin 目录：定义了 P2P 网络插件，这个插件承载了 EOS 系统的 P2P 网络中 TCP/IP 层相关功能，包括建立节点之间握手并互联；监听、发送、接收新交易请求；监听、发送、接收新区块请求；验证接收数据合法性。
- eos/plugins/http_plugin 目录：定义了网络 HTTP 插件，这个插件承载了 EOS 系统的 P2P 网络中 HTTP 层相关功能，包括监听、发送、接收新交易请求；监听、发送、接收新区块请求；验证、接收数据合法性。
- eos/plugins/wallet_plugin 目录（见图 3-6）：定义了钱包插件，这个插件承载了 nodeos 节点程序与钱包交互的相关功能，包括创建、读取钱包文件；设置 unlock timeout 时间；将密钥导入钱包。其中包含的 wallet.cpp 文件实现了对

钱包文件的基本功能，包括创建新钱包、导入密钥等；wallet_manager.cpp 文件实现了对钱包的管理功能，包括设置 unlock 超时时间、lock 指定钱包等；wallet_plugin.cpp 文件对上述功能进行插件化，包括定义插件参数等，实现了 nodeos 节点程序通过调用插件处理钱包文件的功能。

- eos/plugins/acount_history_plugin 目录：定义了历史记录查询插件，该插件承载了节点程序对本地链发起查询的相关功能，包括指定区块查询、指定账户状态查询、指定交易查询。

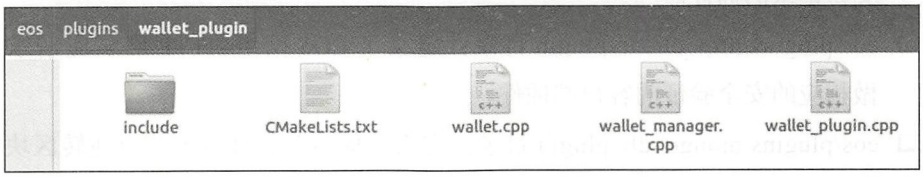


图 3-6　wallet_plugin 文件夹

3. 派生类插件

EOS 项目的插件体系中共有 8 个派生类插件，这些插件分别继承几个基类插件，实现指定功能，如图 3-7 所示。

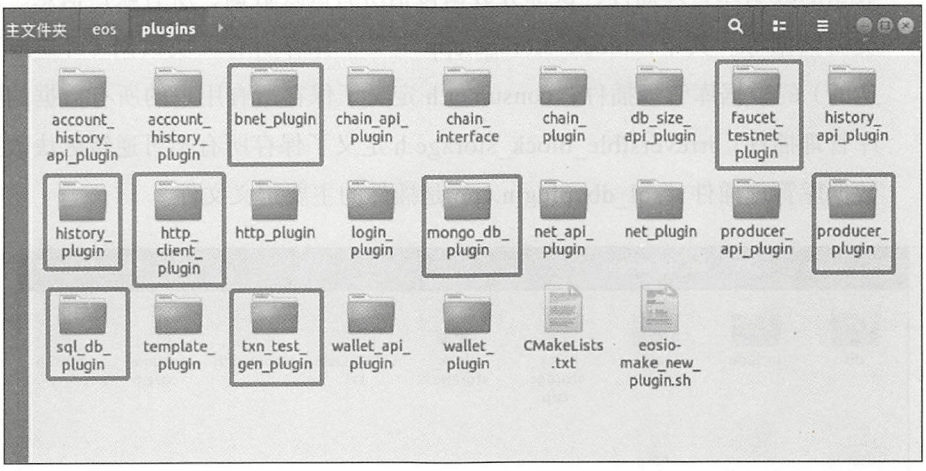


图 3-7　派生类相关文件夹

上图中：

- eos/plugins/bnet_plugin 目录：定义了 EOS 的 P2P 网络中不同节点之间同步各自本地链数据的算法，简单来讲，包括如下特征：查找本地链上的最后一个区块 ID；如果本地产生新区块，则将该区块发送给其他节点；如果本地不产生新区块，则将接收的未确认交易发送给其他节点。
- eos/plugins/faucet_testnet_plugin 目录：定义了在测试网上快速建立测试账号的插件，该插件为测试提供便利。
- eos/plugins/history_plugin 目录：定义了查询指定账户中指定公钥的所有历史交易记录的插件。
- eos/plugins/http_client_plugin 目录：定义了 EOS 网络 HTTP 层响应请求，并做相应的安全验证的客户端插件。
- eos/plugins/mongo_db_plugin 目录：定义了保存并管理本地不可逆转区块链数据的 MongoDB 数据库基本配置插件。
- eos/plugins/producer_plugin 目录：定义了区块生产节点的功能插件，其中功能包括生产、打包新区块数据；对新区块签名；对接收的区块进行验证，包括区块头合法性、签名合法性和交易合法性。
- eos/plugins/sql_db_plugin 目录（见图 3-8）：定义了 EOS 中保存了区块链所有数据的数据库管理插件，这部分数据包括所有区块数据、所有账户信息、所有交易数据等。其中，block_storage.cpp 定义了保存所有区块数据（可逆或不可逆的）的数据库管理插件；consumer.h 定义了保存所有用户的所有数据的数据库管理插件；irreversible_block_storage.h 定义了保存所有不可逆转区块数据的数据库管理插件；sql_db_plugin.cpp 是插件的主要定义文件。

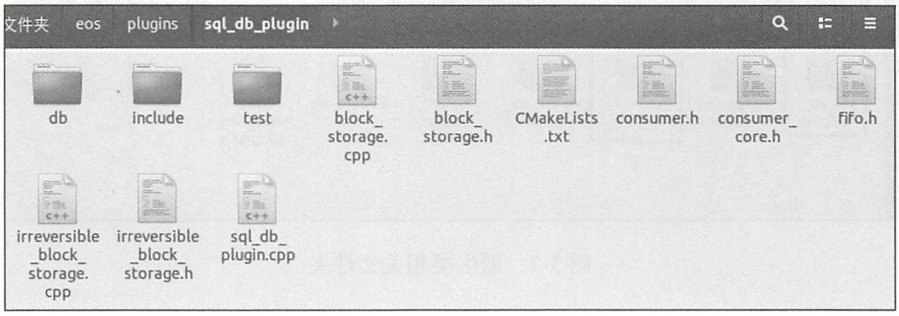


图 3-8 sql_db_plugin 文件夹

- eos/plugins/txn_test_gen_plugin 目录：定义了一个每秒自动产生指定数量的交易信息的插件，该插件主要用作对 EOS 网络的吞吐量（TPS）测试。

4. 封装类插件

该部分插件主要用于为外界与 EOS 链交互提供支持，并对上述几个特定插件进行封装，只暴露 API，如图 3-9 所示。

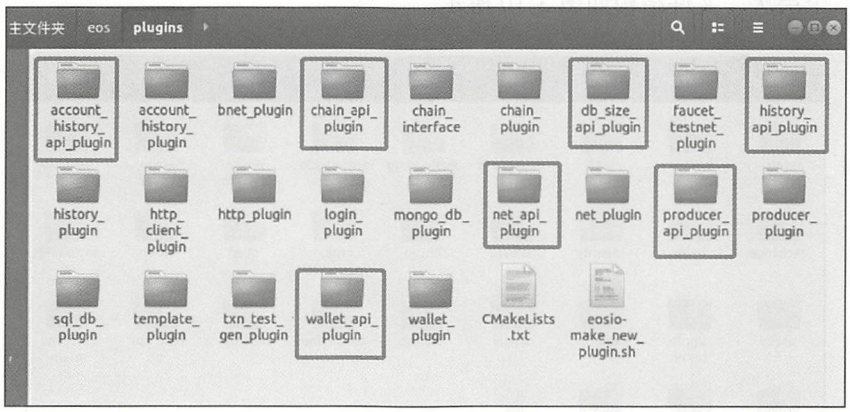


图 3-9　封装类插件

上图中：

- eos/plugins/account_history_api_plugin：实现账户历史记录的接口插件，是与账户历史相关的供外部调用的接口服务。
- eos/plugins/chain_api_plugin：依赖于 chain_plugin，提供与外部调用链相关操作的接口服务。
- eos/plugins/db_size_api_plugin：提供与外部调用及数据有关的接口服务。
- eos/plugins/history_api_plugin：依赖于 history_plugin，提供外部调用查询指定账户中指定公钥的所有历史交易记录的接口服务。
- eos/plugins/net_api_plugin：依赖于 net_plugin，提供与外部调用及网络相关的操作的接口。
- eos/plugins/producer_api_plugin：依赖于 producer_plugin，提供与外部调用及区块生产节点相关功能的接口服务。
- eos/plugins/wallet_api_plugin：依赖于 wallet_plugin，提供与外部调用及钱包交互的接口服务。

仅供非商业用途或交流学习使用

3.1.3 合约层

EOS 项目的基本功能是通过系统合约提供的，用户可以调用已经部署上链的智能合约实现特定的功能，也可以通过 C++ 语言自行编辑 .cpp 智能合约，并通过系统提供的 eosiocpp 编译器将 .cpp、.hpp 文件编译成 .wasm 和 .abi 文件，然后部署上链。EOS 项目的智能合约代码位于 eos/contracts 目录下，包括系统合约层、依赖库层、测试与示例层三个层次。文件结构如图 3-10 所示。

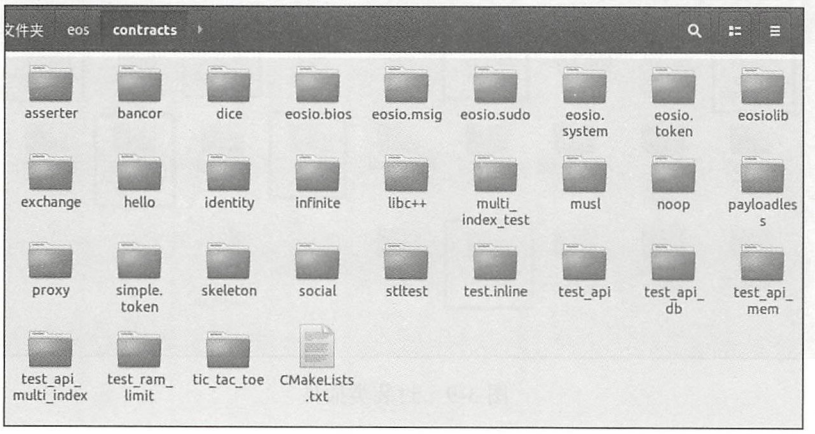


图 3-10 contracts 文件夹

1. 系统合约层

EOS 项目共有 5 个系统级智能合约，如图 3-11 所示。

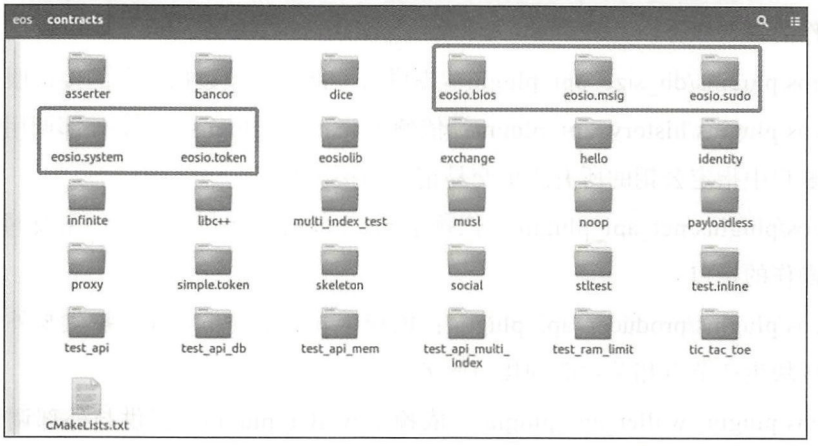


图 3-11 系统合约文件夹

仅供非商业用途或交流学习使用

上图中：

- eos/contracts/eosio.bios：该合约用于启动 EOS 的 P2P 网络。该合约可以直接控制其他账户的资源分配并访问其他特权 API 调用。要启动 P2P 网络，首先，初始启动节点部署该合约，并设置所需的参数；然后，待连接节点通过初始启动节点的地址与之相连；最后，初始启动节点调用 bios 合约，为待连接节点设置权限。
- eos/contracts/eosio.msig 目录：定义了多签系统合约，实现了一个多签功能。由于 EOS 要求系统的每一次更新都需要出块节点完成一次多签，当签名数达到比超级节点的 2/3 多 1 个时更新才能生效，所以出块节点可以调用这个合约实现多签功能。
- eos/contracts/eosio.sudo ：合约实现了创建 EOS 系统中 root 账户的功能，用于修改系统代码与更新合约。
- eos/contracts/eosio.system 目录：可以算是 EOS 系统合约中最重要的部分，实现了 EOS 项目的所有基本功能，如创建新账户、部署智能合约、交易 RAM、抵押获取资源（NET、CPU）、投票和领取节点奖励等。
- eos/contracts/eosio.Token 目录：实现了发行 Token 的功能，可以说 EOS 本身代币或后续基于 EOS 的代币都是通过该合约发行的，主体函数包括发行新代币、初始分发、转账、查询余额。

2. 依赖库层

上述 5 个系统合约的实现离不开一个庞大的依赖库，其中包括数据的定义（account、asset 等），还包括一些常用函数（权限管理、序列化等），如图 3-12 所示。

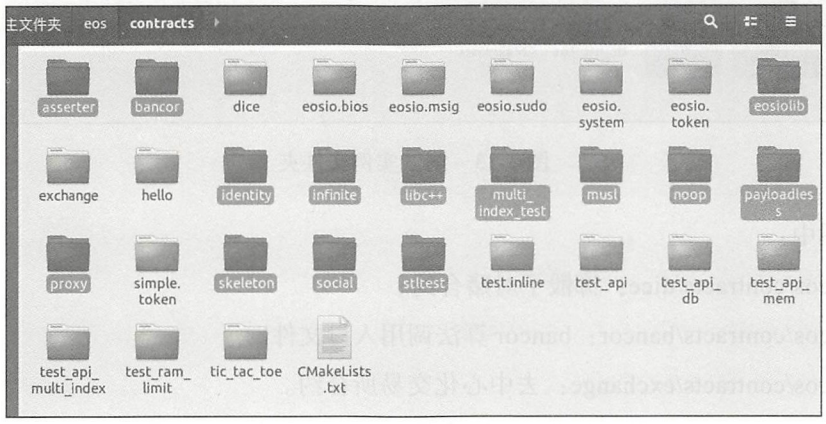


图 3-12　依赖库的文件夹

仅供非商业用途或交流学习使用

上图中：

- eos/contracts/asserter：定义了 assert 的相关结构体，并完成对智能合约事件的分发。
- eos/contracts/bancor：定义了 bancor 结构体，包含与凯恩斯国际货币单位相关的内容，主要是货币单位之间的转换。
- eos/contracts/ eosiolib：包含 EOS 运行所依赖的库的头文件。
- eos/contracts/ musl：Linux 操作系统下的一个标准库。
- eos/contracts/ noop：实现一个空的智能合约。
- eos/contracts/ proxy：实现代理的相关内容。

3. 测试与示例层

EOS 给出了一些有趣的示例合约和测试文件供用户理解原理与测试功能，如图 3-13 所示。

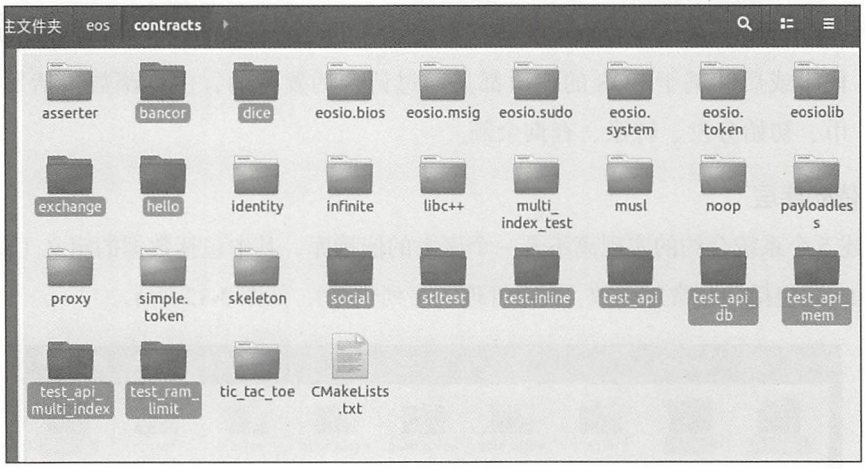


图 3-13 测试实例文件夹

上图中：

- eos/contracts/dice：掷骰子对赌合约。
- eos/contracts/bancor：bancor 算法调用入口文件。
- eos/contracts/exchange：去中心化交易所合约。
- eos/contracts/hello：helloworld 合约。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- eos/contracts/social：类似 Steem 的社交平台合约（只包含基本功能）。
- eos/contracts/test_ 前缀：一些测试文件。

3.1.4　支持层

支持层面涉及的内容就比较多了，它包括主程序模块、依赖库模块、测试模块、Docker 工具模块、文档说明模块。下面我们分别对这些模块进行一一分析。

1. 主程序模块

EOS 项目的主程序源代码位于 eos/program 目录下，包含 6 个基本功能组件，如图 3-14 所示。

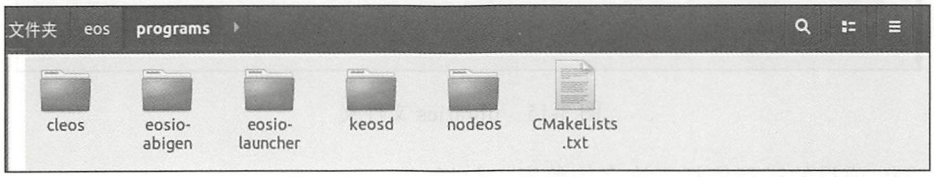


图 3-14　program 文件夹

上图中：

- nodeos：EOS 项目的核心程序源代码，可以配置不同插件来运行不同类型节点。该进程主要负责提供区块生产，封装 API 接口和本地开发的功能。
- cleos：EOS 项目中与 nodeos 之间交互信息的命令行工具 cleos 的源代码，编译后与 nodeos 公开的 REST API 进行信息交互。
- keosd：EOS 项目中的内置钱包程序，配合钱包相关插件通过 HTTP 接口或 RPC API 与 cleos 进行通信，完成账户创建、转账等基本钱包功能。
- eosio-abigen：EOS 项目的智能合约编译器源代码（用于生成智能合约的 .abi 文件）。
- eosio-applesedemo：EOS 项目中公私钥生成与数字签名生成组件的源代码，主要为 EOS 项目提供密码学（采用 ECDSA 算法中的 secp256r1）支持，其中 main.cpp 为该工具的核心主函数、r1_signature_compactor 为数字签名功能描述文件、sign.sh 为生成应用程序包 applesedemo.app 并对其进行签名的自动化脚本。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- eosio-launcher：EOS 项目的 P2P 网络组成启动器源代码，该部分简化了 nodeos 节点组网的流程。

2. 依赖库模块

EOS 项目的运行依托于许多依赖库，这些依赖库的基本信息位于 eos/libraries 目录中，如图 3-15 所示。

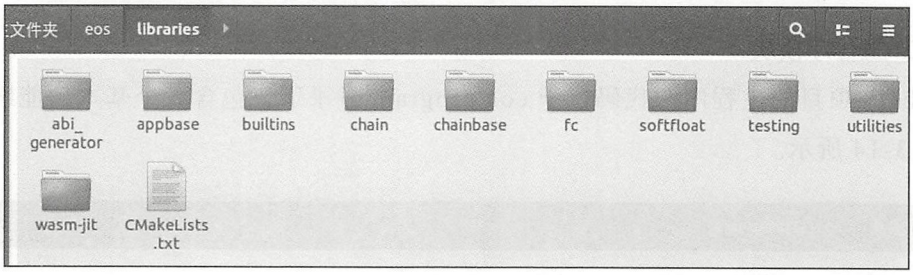


图 3-15 libraries 文件夹

EOS 项目的运行依托于许多依赖库，包括：

- abi_generator：包含了智能合约编译器所需的依赖文件，可以说编译器的主要代码就是位于这个目录中。
- appbase：提供了一个用于从一组插件构建应用程序的基本框架。该模块负责管理插件的生命周期，并确保所有插件按正确的顺序进行配置、初始化、启动和关闭。该依赖库包含 5 个主要特征，即动态指定要加载的插件、自动加载依赖插件、插件可以指定命令行参数和配置文件选项、程序正常退出 SIGINT 和 SIGTERM、最小依赖（Boost 1.60，C++ 14）。
- builtins：包含了 EOS 项目编译过程中所需要的 compiler-RT 编译器（libgcc 的替换库）的源代码，包括编译器本身以及相关功能函数的代码描述。
- chain：EOS 项目的核心内容，包括区块、区块链、merkle 树等数据结构以及初始区块、控制器等关键算法。
- chainbase：定义了保存 EOS 区块链数据的数据库结构，可以说该数据库旨在满足区块链应用程序的苛刻要求，以使其适用于任何需要强大的事务数据库的应用程序。
- fc：EOS 项目的细胞级模块，定义了 EOS 项目中的基本变量数据结构，包括 String、Time、Base 系列编码等。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- softfloat：包含了一个 Berkeley SoftFloat，即符合 IEEE 浮点运算标准的二进制浮点软件实现。
- testing：包含了几个测试实例，包括对区块链数据库的链接测试、P2P 网络的链接测试等。
- utilities：主要包含了一些通用的标准函数。
- wasn-jit：包含了一个 WebAssembly 的独立 VM。它可以加载标准的二进制格式，也可以加载 WebAssembly 参考解释器定义的文本格式。对于文本格式，它可以加载标准堆栈机器语法和参考解释器使用的老式 AST 语法，以及所有测试命令。

3. 测试模块

EOS 提供了一些测试文件，供用户测试节点是否运行正常，其中测试分两大类：

- 对链功能的测试，包括与区块链之间的数据交互、transaction 分发等。
- 对网络层的测试，包括 P2P 网络传输功能、cleos 与 nodeos 之间的通信等。

测试模块主要位于 eos/tests 文件夹，如图 3-16 所示。

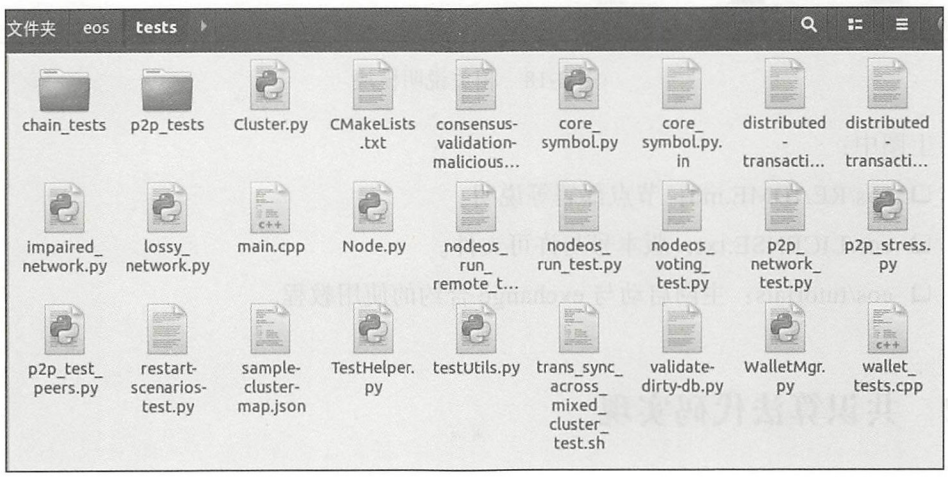


图 3-16　tests 文件夹

4. Docker 工具模块

EOS 允许通过 Docker 运行节点或钱包，eos/Docker 文件夹下包含了一些通过 Docker 启动并运行程序的必要文件，如图 3-17 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

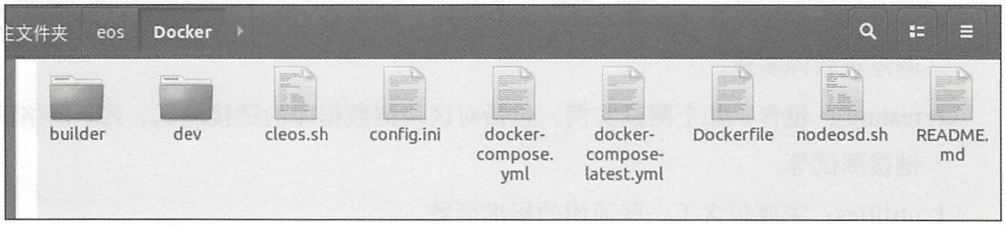


图 3-17 Docker 文件夹

5. 文档说明模块

文档说明模块如图 3-18 所示。

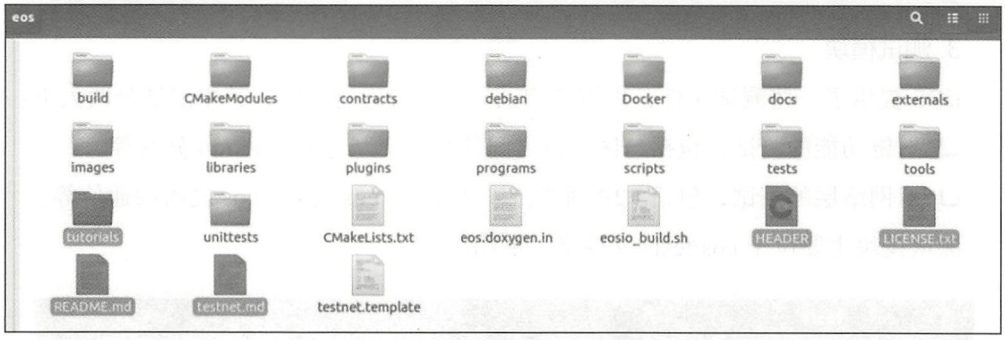


图 3-18 相关说明模块

上图中：

- eos/README.md：节点部署等说明。
- eos/LICENSE.txt：版本号与许可文件。
- eos/tutorials：主网启动与 exchange 合约的使用教程。

3.2 共识算法代码实现

拜占庭位于如今土耳其的伊斯坦布尔，是东罗马帝国的首都。由于当时拜占庭罗马帝国国土辽阔，出于防御目的，每个军队都分隔很远，将军与将军之间只能靠信差传递消息。在战争时期，拜占庭军队内所有将军和副官必须达成一致的共识，都认为有赢的机会才去攻打敌人的阵营。但是，在军队内有可能存有叛徒和敌军的间谍，他们会左右将军们的决定又扰乱整体军队的秩序。将军们采用投票的策略来决定是进攻

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

还是撤退，也就是说如果多数人决定进攻，就冲上去；如果多数人决定撤退，就撤退。

这时候，在已知有成员谋反的情况下（或者是有奸细的情况下，他们乱投票，或者擅自修改军令），忠诚的将军在不受叛徒的影响下如何达成一致的协议？这是一个问题，也就是拜占庭将军问题。拜占庭将军问题是一个协议问题，拜占庭帝国军队的将军们必须全体一致的决定是否攻击某一支敌军。问题是这些将军在地理上是分隔开来的，并且将军中存在叛徒。叛徒可以采取任意行动以达到以下目标：欺骗某些将军采取进攻行动；促成一个不是所有将军都同意的决定，如当将军们不希望进攻时促成进攻行动；或者迷惑某些将军，使他们无法做出决定。如果叛徒达到了这些目的中的任意一个，则任何攻击行动的结果都是注定要失败的，只有完全达成一致的努力才能获得胜利。所以，拜占庭将军问题是一个分布式系统中的共识问题，也是一个分布式容错系统的问题。拜占庭将军问题不是说要让将军达成进攻的决定，而是让所有忠诚将军达成共识，要么全都进攻，要么全都撤退，不至于被敌方各个攻破。

这种共识问题在区块链系统中也是存在的，目前也已经有许多的方法进行解决了，其中一种解决办法就是共识算法。所谓的共识算法，简单来说就是在一个不可信网络中，通过大家投票的方式来对某个确定性结果达成一致，这是一种少数服从多数的方法。目前在区块链系统中，PBFT 和 Raft 是联盟链和私有链上常用的共识算法，而在公有链上，常用的是 PoW、PoS、DPoS 等共识算法。不管是 PoW、PoS 还是 DPoS 共识算法，本质上都是一种投票方法。EOS 之所以有可能达成百万 TPS 并且不会有分叉的性能问题，主要是因为 BM 选择了基于授权证明 + 拜占庭容错的共识算法（DPoS+BFT）。

3.2.1　DPoS 机制实现关键代码分析

我们先来简单理解一下为什么 DPoS 相比于 PoW 算法性能更优。我们通过类比古希腊公民大会制度与西方议会制度来说明这个问题。古希腊公民大会制度概念：（ecclesia，亦可写作 ekklesia）是古希腊城邦和古罗马的最高权力机关。由王或议事会召集，全体成年男子（战时全体战士）参加，讨论、决定部落各项重大问题。通常用举手或喊声表决，并达成对国家重大事务的共识。在这里可以看出其优点很明显——更加民主。但是缺点亦更加明显，共识效率是这种制度最大的钳制。对于小国寡民的情况，这种共识模型看起来很美好，很民主。但是对于一个大国，这种制度就有了很

非卖品，仅供非商业用途或交流学习使用

大弊端，任何国家事务（事无巨细）都要组织少至千万，多至几亿的国民参与投票，效率何其低下。

议员制度概念：这个概念大家都很了解，通过民主选举产生，对公民负责并受公民监督的各级议员，各级议员负责各级事物的讨论决策。优点在于这种制度极大地提升了共识的效率，通过民主投票的方式选举出一组代表公民参与最终的决策，使得最终决策的效率得到极大提高（参与共识的人数变小，单单看统计票数的工作都能减小不少的工作量），不可避免的，这种模式中缺点也是存在的，一定程度上降低了民主程度。

类比区块链世界的共识算法，优缺点何其相似。PoW 算法就相当于古希腊公民大会制度，具有很高的去中心化程度，但是共识效率比较低；DPoS 算法就相当于议员制度，提高了共识效率，但是相对弱化了去中心化程度，是以牺牲绝对的去中心化来换取共识效率。两者间的关系如图 3-19 和图 3-20 所示。

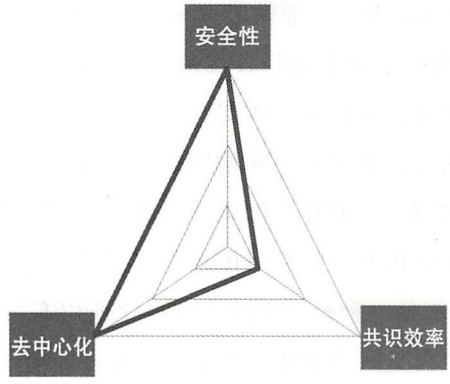


图 3-19 PoW 共识算法

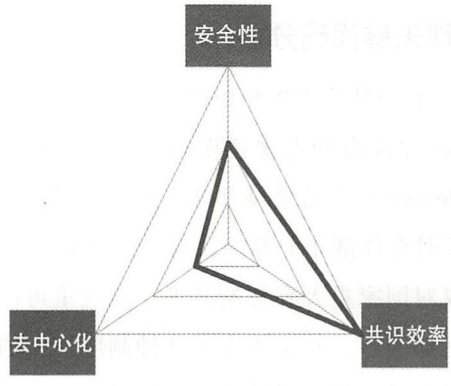


图 3-20 DPoS 共识算法

非卖品，仅供非商业用途或交流学习使用

EOS 项目中整个出块节点竞选（DPoS）算法是依托系统合约实现的，简单来说，整个 DPoS 算法可以简化为两个进程——投票与更新、监控票数后出块。图 3-21 为出块节点竞选类图。

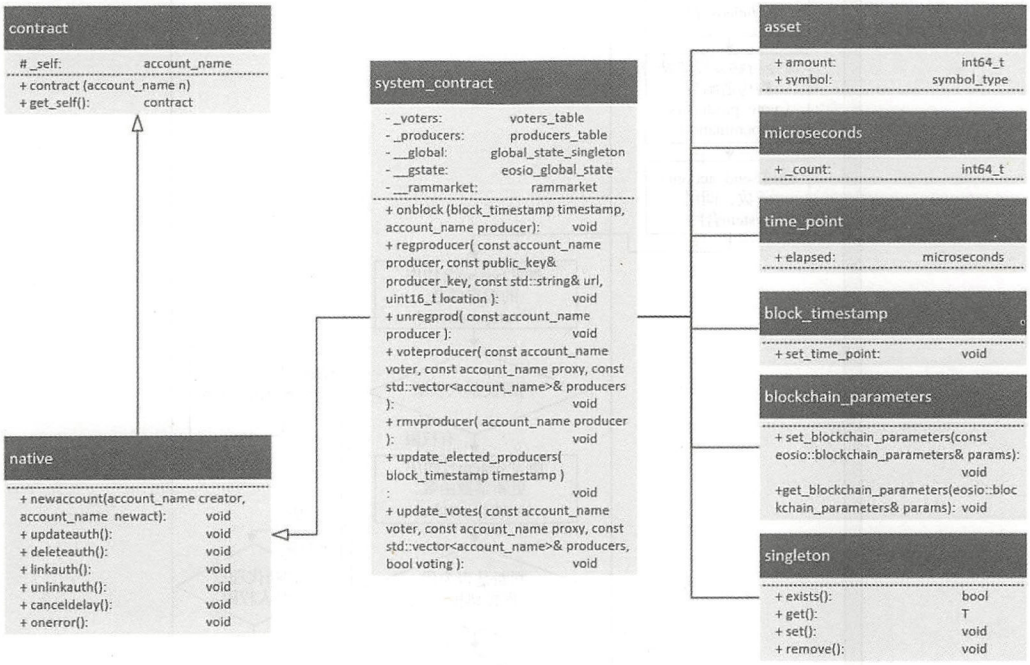


图 3-21　出块节点竞选类图

我们主要来看一下投票的总流程图，如图 3-22 所示，其中关键文件有两个：

- programs/cleos/main.cpp 文件是 cleos 的源程序，主要负责投票信息的捕获。
- contracts/eosio.system/voting.cpp 文件，将由上述 cleos 源程序中捕获的投票信息送入 eosio.system 合约中，触发相应的函数来进行投票，eosio.system 合约中关于投票函数的实现位于 voting.cpp 中。

programs/cleos/main.cpp 定义了 cleos 的源程序。cleos 是一个 EOS 命令行工具，为用户提供一种通过命令行与区块链交互并管理钱包的功能。其中涉及与出块节点竞选相关的代码主要有四部分（注册、注销节点；投票或代理投票；增删出块节点；查询出块节点）：

- register_producer_subcommand 结构：注册出块节点。

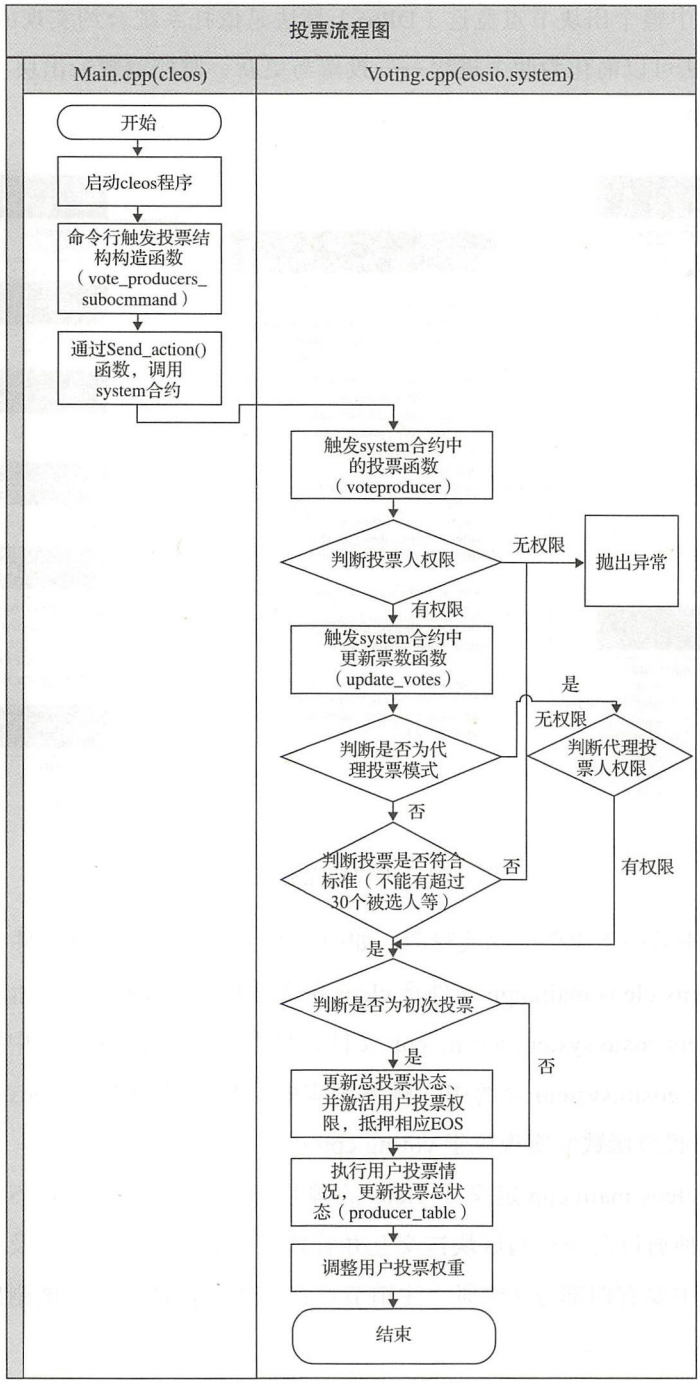


图 3-22 出块节点竞选总流程图

- unregister_producer_subcommand 结构：注销出块节点。
- vote_producer_proxy_subcommand 结构：通过代理节点投票。
- vote_producers_subcommand 结构：执行投票操作。
- approve_producer_subcommand 结构：增加待选出块节点。
- unapprove_producer_subcommand 结构：取消待选出块节点。
- list_producers_subcommand 结构：查询可选出块节点列表。

cleos 中指令的执行是通过在 main 函数中声明相应的结构实现的，每个结构中包含具体的构造函数，用以执行相应的功能逻辑。首先我们来看一下 main 函数中关于 vote 动作的定义。由于 main 函数很长，我们只选取其中 voting 部分的代码，如下所示：

```
auto voteProducer = system->add_subcommand("voteproducer", localized("Vote
    for a producer"));
    voteProducer->require_subcommand();
    auto voteProxy = vote_producer_proxy_subcommand(voteProducer);
    auto voteProducers = vote_producers_subcommand(voteProducer);
    auto approveProducer = approve_producer_subcommand(voteProducer);
    auto unapproveProducer = unapprove_producer_subcommand(voteProducer);
    auto listProducers = list_producers_subcommand(system);
```

简单来说，main 函数主要通过 add_subcommand() 函数赋予的定义 cleos 工具主要选项的功能初始化 voteproducer 功能，然后通过 app.parse(argc, argv) 函数捕获用户的输入，并调用相关的功能。app.parse() 函数的代码如下所示：

```
try {
        app.parse(argc, argv);
    } catch (const CLI::ParseError &e) {
        return app.exit(e);
    } catch (const explained_exception& e) {
        return 1;
    } catch (connection_exception& e) {
        if (verbose_errors) {
            elog("connect error: ${e}", ("e", e.to_detail_string()));
        }
    } catch (const fc::exception& e) {
        // 如果存在错误尝试获取
        if (!print_recognized_errors(e, verbose_errors)) {
            // 没有识别错误
            if (!print_help_text(e) || verbose_errors) {
```

```
                elog("Failed with error: ${e}", ("e", verbose_errors ? e.to_
                    detail_string() : e.to_string()));
            }
        }
        return 1;
    }
```

当捕获到“system+voteproducer+ 详细参数”命令时，触发 vote_producers_subcommand 结构中的构造函数。构造函数首先初始化子命令，而后执行 send_actions() 函数，将获取的参数发送到系统合约（eosio.system）中，并将捕获的智能合约的执行结果返回给用户。以下为 vote 函数代码：

```
struct vote_producers_subcommand {
    string voter_str;
    vector<eosio::name> producer_names;

    vote_producers_subcommand(CLI::App* actionRoot) {
        auto vote_producers = actionRoot->add_subcommand("prods", localized("Vote
            for one or more producers"));
        vote_producers->add_option("voter", voter_str, localized("The voting
            account"))->required();
        vote_producers->add_option("producers", producer_names, localized("The
            account(s) to vote for. All options from this PoSition and
            following will be treated as the producer list."))->required();
        add_standard_transaction_options(vote_producers);

        vote_producers->set_callback([this] {

            std::sort( producer_names.begin(), producer_names.end() );

            fc::variant act_payload = fc::mutable_variant_object()
                ("voter", voter_str)
                ("proxy", "")
                ("producers", producer_names);
            send_actions({create_action({permission_level{voter_str,config::
                active_name}}, config::system_account_name, N(voteproducer),
                act_payload)});
        });
    }
};
```

具体投票的代码是由系统合约负责执行的，这部分代码实现位于 contracts/eosio.system/voting.cpp 中、首先我们罗列一下其中的关键函数并介绍一下其含义与功能：

- regproducer 函数：注册出块节点账户。
- unregprod 函数；注销出块节点账户。
- update_elected_producers：更新出块节点选举情况的全局状态。
- stake2vote 函数：为投票抵押代币。
- voteproducer：投票函数。
- update_votes：更新投票结果。
- regproxy：注册委托投票代理人。

系统合约在接到 cleos 中捕获的投票命令之后，将会首先触发 voteproducer 函数，这一步是为了判断投票人是否具有投票权限。若具备，则执行 update_votes 函数以更新全局投票情况；反之则抛出异常。voteproducer() 函数的代码如下所示：

```
void system_contract::voteproducer( const account_name voter_name, const
    account_name proxy, const std::vector<account_name>& producers ) {
    require_auth( voter_name );
    update_votes( voter_name, proxy, producers, true );
}
```

接着执行 update_votes() 函数，下面我们做详细介绍：

（1）判断投票人类型，若为代理投票人，则将判断代理人是否具有对应的委托权限，不符则抛出异常，相符则向下继续执行。

（2）判断投票人是否具有投票权限，即是否进行用以投票的代币抵押。这里主要查询一个全局表 _voters，该表中维护了所有用户的抵押信息。若没有抵押，则抛出异常；通过则向下继续执行。

（3）判断本次投票是否为首次投票。若是，则修改全局投票状态 _gstate.total_activated_stake；接着进行投票操作，即读取用户提交的 producer_deltas 列表，并在出块节点的全局表中查找对应出块节点，并调整相应节点的投票情况。

（4）修改该用户的投票权重。

```
void system_contract::update_votes( const account_name voter_name, const
    account_name proxy, const std::vector<account_name>& producers, bool
    voting ) {
    //验证输入
    if ( proxy ) {
        eosio_assert( producers.size() == 0, "cannot vote for producers and
            proxy at same time" );
```

```
    eosio_assert( voter_name != proxy, "cannot proxy to self" );
    require_recipient( proxy );
} else {
    eosio_assert( producers.size() <= 30, "attempt to vote for too many
        producers" );
    for( size_t i = 1; i < producers.size(); ++i ) {
        eosio_assert( producers[i-1] < producers[i], "producer votes
            must be unique and sorted" );
    }
}

auto voter = _voters.find(voter_name);
eosio_assert( voter != _voters.end(), "user must stake before they can
    vote" ); /// 投票前需先抵押
eosio_assert( !proxy || !voter->is_proxy, "account registered as a proxy
    is not allowed to use a proxy" );

/**
 * The first time someone votes we calculate and set last_vote_weight,
     since they cannot unstake until
 * after total_activated_stake hits threshold, we can use last_vote_
     weight to determine that this is
 * their first vote and should consider their stake activated.
 */
if( voter->last_vote_weight <= 0.0 ) {
    _gstate.total_activated_stake += voter->staked;
    if( _gstate.total_activated_stake >= min_activated_stake && _gstate.
       thresh_activated_stake_time == 0 ) {
        _gstate.thresh_activated_stake_time = current_time();
    }
}

auto new_vote_weight = stake2vote( voter->staked );
if( voter->is_proxy ) {
    new_vote_weight += voter->proxied_vote_weight;
}

boost::container::flat_map<account_name, pair<double, bool /*new*/> >
    producer_deltas;
if ( voter->last_vote_weight > 0 ) {
    if( voter->proxy ) {
        auto old_proxy = _voters.find( voter->proxy );
        eosio_assert( old_proxy != _voters.end(), "old proxy not found"
            ); //data corruption
```

```
            _voters.modify( old_proxy, 0, [&]( auto& vp ) {
                vp.proxied_vote_weight -= voter->last_vote_weight;
            });
            propagate_weight_change( *old_proxy );
        } else {
            for( const auto& p : voter->producers ) {
                auto& d = producer_deltas[p];
                d.first -= voter->last_vote_weight;
                d.second = false;
            }
        }
    }

    if( proxy ) {
        auto new_proxy = _voters.find( proxy );
        eosio_assert( new_proxy != _voters.end(), "invalid proxy specified" );
           //if ( !voting ) { data corruption } else { wrong vote }
        eosio_assert( !voting || new_proxy->is_proxy, "proxy not found" );
        if ( new_vote_weight >= 0 ) {
            _voters.modify( new_proxy, 0, [&]( auto& vp ) {
                vp.proxied_vote_weight += new_vote_weight;
            });
        propagate_weight_change( *new_proxy );
        }
    } else {
        if( new_vote_weight >= 0 ) {
            for( const auto& p : producers ) {
                auto& d = producer_deltas[p];
                d.first += new_vote_weight;
                d.second = true;
            }
        }
    }

    for( const auto& pd : producer_deltas ) {
        auto pitr = _producers.find( pd.first );
        if( pitr != _producers.end() ) {
        eosio_assert( !voting || pitr->active() || !pd.second.second /* not
            from new set */, "producer is not currently registered" );
        _producers.modify( pitr, 0, [&]( auto& p ) {
            p.total_votes += pd.second.first;
            if ( p.total_votes < 0 ) { // floating point arithmetics can give
                small negative numbers
                p.total_votes = 0;
```

```
            }
            _gstate.total_producer_vote_weight += pd.second.first;
            //eosio_assert( p.total_votes >= 0, "something bad happened" );
        });
    } else {
        eosio_assert( !pd.second.second /* not from new set */, "producer is
            not registered" ); //data corruption
        }
    }

    _voters.modify( voter, 0, [&]( auto& av ) {
        av.last_vote_weight = new_vote_weight;
        av.producers = producers;
        av.proxy     = proxy;
    });
}
```

至此，投票操作已经完成，但是请注意，在代码中并没有涉及对节点更新修改后的全局变量值进行的操作。其实更新操作并非在 cleos 程序中进行的，而是在 nodeos 程序中通过插件调取 update_elected_producers 函数来完成的：每过一个区块，或者是区块时间戳每更新一次，就执行一次该函数，并完成全局变量的更新。

系统将会在 top_producers 表中保留前 21 个节点信息，使其成为出块节点，并调用 std::sort() 函数对这些节点进行排序，而后被选出来的出块节点之间进行共识的 BFT 算法就是参照这个排序顺序进行的。update_elected_producers() 函数代码如下所示：

```
void system_contract::update_elected_producers( block_timestamp block_time ) {
    _gstate.last_producer_schedule_update = block_time;

    auto idx = _producers.get_index<N(prototalvote)>();

    std::vector< std::pair<eosio::producer_key,uint16_t> > top_producers;
    top_producers.reserve(21);

    for ( auto it = idx.cbegin(); it != idx.cend() && top_producers.size() <
       21 && 0 < it->total_votes && it->active(); ++it ) {
        top_producers.emplace_back( std::pair<eosio::producer_key,uint16_
            t>({{it->owner, it->producer_key}, it->location}) );
    }

    if ( top_producers.size() < _gstate.last_producer_schedule_size ) {
        return;
```

仅供非商业用途或交流学习使用

```
    }

    ///根据出块节点名排序
    std::sort( top_producers.begin(), top_producers.end() );

    std::vector<eosio::producer_key> producers;

    producers.reserve(top_producers.size());
    for( const auto& item : top_producers )
        producers.push_back(item.first);

    bytes packed_schedule = pack(producers);

    if( set_proPoSed_producers( packed_schedule.data(),   packed_schedule.
       size() ) >= 0 ) {
        _gstate.last_producer_schedule_size  =  static_cast<decltype(_gstate.
            last_producer_schedule_size)>( top_producers.size() );
    }
}
```

3.2.2　BFT 算法实现关键代码分析

EOS 项目中整个超级节点之间生产区块所采用的共识算法为普通的拜占庭容错算法（BFT），其类图如图 3-23 所示。该算法整体依托插件体系实现，之所以采用这样的设计模式，主要有两层原因：防止部分恶意超级节点串通作恶，包括修改账本信息等动作；防止区块链分叉。

我们来看一下区块生产的总流程，如图 3-24 所示，其中的关键文件有 4 个：

- programs/nodeos/main.cpp 文件：该文件是 nodeos 的源程序，主要负责节点出块与接收区块等。
- plugins/producer_plugin/producer_plugin.cpp 文件：这个插件主要负责与节点有关的生产区块、接收区块和同步区块的所有函数。
- libraries/appbase/channel.cpp 文件：该文件定义了 EOS 系统中的 transaction、block 等数据的交互通道。
- plugins/chain_plugin/chain_plugin.cpp 文件：这个插件主要负责与 EOS 区块链数据之间的交互操作，如读取、写入、更新、删除等。

仅供非商业用途或交流学习使用

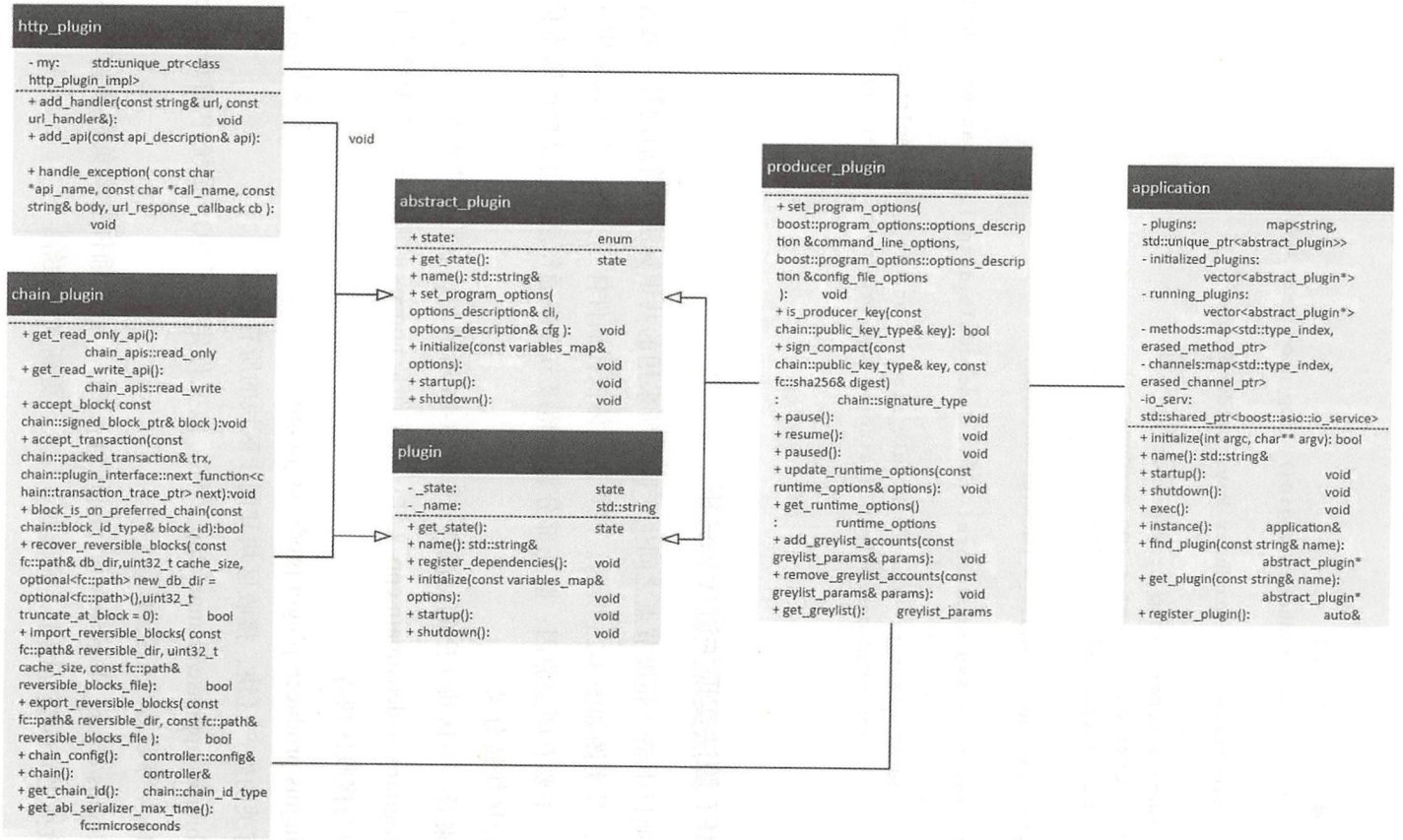


图 3-23　生产区块的 BFT 算法类图

仅供非商业用途或交流学习使用

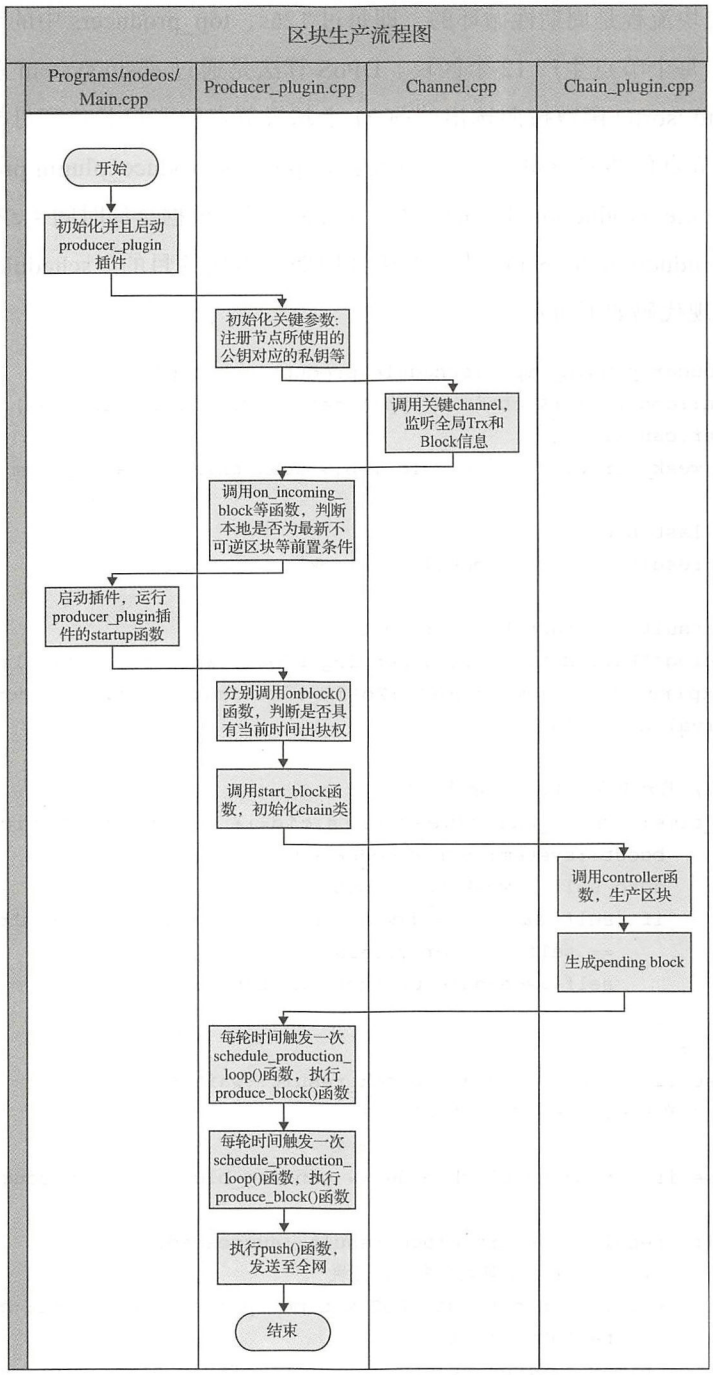


图 3-24　出块节点竞选总流程图

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

EOS 的出块流程是周期性循环的，即每过 126s，top_producers 中的节点即重复一次出块流程，每个节点生产 12 个区块。DPoS 算法是通过 eosio.system 系统合约实现的，并根据 std::sort() 函数按照选出的前 21 个超级节点的账户名首字母排序，而控制这 21 个出块节点的生产区块的具体函数就是 plugins/produce_plugin/produce_plugin.cpp 中的 schedule_production_loop() 函数：nodeos 程序根据全局时间（ePoCh）重复执行 schedule_production_loop() 函数，以达到周期性出块的目的。schedule_production_loop() 函数实现代码如下所示：

```
void producer_plugin_impl::schedule_production_loop() {
    chain::controller& chain = app().get_plugin<chain_plugin>().chain();
    _timer.cancel();
    std::weak_ptr<producer_plugin_impl> weak_this = shared_from_this();

    bool last_block;
    auto result = start_block(last_block);

    if (result == start_block_result::failed) {
        elog("Failed to start a pending block, will try again later");
_timer.expires_from_now( boost::PoSix_time::microseconds( config::block_
    interval_us  / 10 ));

        // 启动节点失败，过会继续尝试
        _timer.async_wait([weak_this,cid=++_timer_corelation_id](const
            boost::system::error_code& ec) {
            auto self = weak_this.lock();
            if (self && ec != boost::asio::error::operation_aborted && cid
                == self->_timer_corelation_id) {
                self->schedule_production_loop();
            }
        });
    } else if (result == start_block_result::waiting) {
        // 在更多的区块来之前没有操作可做

    } else if (_pending_block_mode == pending_block_mode::producing) {

        if (result == start_block_result::succeeded) {
            // 不迟于截止日期之前发送此区块
            static const boost::PoSix_time::ptime ePoCh(boost::gregorian::da
                te(1970, 1, 1));
            _timer.expires_at(ePoCh + boost::PoSix_time::microseconds(chain.
                pending_block_time().time_since_ePoCh().count() + (last_
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
                block ? _last_block_time_offset_us : _produce_time_offset_
                us)));
            fc_dlog(_log, "Scheduling Block Production on Normal Block
                #${num} for ${time}", ("num", chain.pending_block_state()-
                >block_num)("time",chain.pending_block_time()));
        } else {
            // 立刻发送该区块
            _timer.expires_from_now( boost::PoSix_time::microseconds( 0 ));
            fc_dlog(_log, "Scheduling Block Production on Exhausted Block
                #${num} immediately", ("num", chain.pending_block_state()-
                >block_num));
        }

_timer.async_wait([&chain,weak_this,cid=++_timer_corelation_id](const
    boost::system::error_code& ec) {
            auto self = weak_this.lock();
            if (self && ec != boost::asio::error::operation_aborted && cid
                == self->_timer_corelation_id) {
                auto res = self->maybe_produce_block();
                fc_dlog(_log, "Producing Block #${num} returned: ${res}",
                    ("num", chain.pending_block_state()->block_num)("res",
                    res) );
            }
        });
    } else if (_pending_block_mode == pending_block_mode::speculating && !_
        producers.empty() && !production_disabled_by_policy()){
            // 如果我们有任何生产者，那么我们至少应该为下一个可用插槽设置一个计时器
            optional<fc::time_point> wake_up_time;
            for (const auto&p: _producers) {
                auto next_producer_block_time = calculate_next_block_time(p);
                if (next_producer_block_time) {
                    auto producer_wake_up_time = *next_producer_block_time -
                        fc::microseconds(config::block_interval_us);
                        if (wake_up_time) {
                            // 判断当前是否处于唤醒时间点
                            wake_up_time = std::min<fc::time_point>(*wake_
                                up_time, producer_wake_up_time);
                        } else {
                            wake_up_time = producer_wake_up_time;
                        }
                    }
            }

            if (wake_up_time) {
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
                fc_dlog(_log, "Specualtive Block Created; Scheduling
                    Speculative/Production Change at ${time}", ("time",
                    wake_up_time));
                static const boost::PoSix_time::ptime ePoCh(boost::gregorian
                    ::date(1970, 1, 1));
                _timer.expires_at(ePoCh + boost::PoSix_time::microseconds(wake_
                    up_time->time_since_ePoCh().count()));
                _timer.async_wait([weak_this,cid=++_timer_corelation_id](const
                    boost::system::error_code& ec) {
                    auto self = weak_this.lock();
                    if (self && ec != boost::asio::error::operation_aborted
                        && cid == self->_timer_corelation_id) {
                        self->schedule_production_loop();
                    }
                });
            } else {
                fc_dlog(_log, "Speculative Block Created; Not Scheduling
                    Speculative/Production, no local producers had valid
                    wake up times");
            }
        } else {
            fc_dlog(_log, "Speculative Block Created");
        }
    }
```

总结下来，该函数的主要逻辑如下：通过 chain_plugin 插件调用 controller 函数，并重置时间；调用 start_block() 函数，尝试打包新区块，返回 start_block_result 状态（该状态是 enum 枚举类，共有四种状态：succeeded、failed、waiting、exhausted）。

根据返回的类型针对性地执行函数：

- failed：提示出块错误消息，挂起至同步状态；
- waiting：不做任何操作，保持同步状态；
- succeeded：出块成功，若生产的区块不是 exhausted 状态（超过规定的出块时间），则调用 maybe_produce_block 函数尝试生产并发布区块；判断该节点是否有其他出块账户，若有，等待下次出块。监听出块节点动态，判断自己是否处在下一轮出块列表中。通过这样的设置，nodeos 节点就可以持续跟踪自己是否具有当前 ePoCh 的出块权限，若具有，则出块，反之则同步。其中有一个关键函数——start_block() 函数，下面对其展开介绍。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

start_block() 函数是 EOS 中生产区块的函数，通过这个函数可以保证节点在处于出块环节时生产区块。首先来看一下 start_block() 函数的源码：

```
void start_block( block_timestamp_type when, uint16_t confirm_block_count,
    controller::block_status s ) {
        EOS_ASSERT( !pending, block_validate_exception, "pending block is
            not available" );

        EOS_ASSERT( db.revision() == head->block_num, database_exception,
            "db revision is not on par with head block",
                ("db.revision()", db.revision())("controller_head_block", head-
                    >block_num)("fork_db_head_block", fork_db.head()->block_
                    num) );

        auto guard_pending = fc::make_scoped_exit([this](){
            pending.reset();
        });

        pending = db.start_undo_session(true);

        pending->_block_status = s;

        pending->_pending_block_state = std::make_shared<block_state>(
            *head, when ); //如果存在待处理计划，则将其激活
        pending->_pending_block_state->in_current_chain = true;

        pending->_pending_block_state->set_confirmed(confirm_block_count);

        auto was_pending_promoted = pending->_pending_block_state->maybe_
            promote_pending();

        //只有当我们推测读取模式时才修改推测块中的状态（否则我们需要清空头部或不可逆读取的状态）
        if ( read_mode == db_read_mode::SPECULATIVE || pending->_block_
            status != controller::block_status::incomplete ) {

            const auto& gpo = db.get<global_property_object>();
            if( gpo.proPoSed_schedule_block_num.valid() &&
                ( *gpo.proPoSed_schedule_block_num <= pending->_pending_
                    block_state->DPoS_irreversible_blocknum ) &&
pending->_pending_block_state->pending_schedule.producers.size() == 0 &&
    // ... 还有一个新的待定时间表的空间 ...
                !was_pending_promoted // ...不只是因为它在这个区块的开头被提升为活跃的
        )
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
{
    // 将提议的时间表推广到待定时间表
    if( !replaying ) {
        ilog( "promoting proPoSed schedule (set in block
            ${proPoSed_num}) to pending; current block: ${n} lib:
            ${lib} schedule: ${schedule} ",
            ("proPoSed_num", *gpo.proPoSed_schedule_block_num)
                ("n", pending->_pending_block_state->block_num)
            ("lib", pending->_pending_block_state->DPoS_
                irreversible_blocknum)
            ("schedule", static_cast<producer_schedule_type>
                (gpo.proPoSed_schedule) ) );
    }
    pending->_pending_block_state->set_new_producers( gpo.proPoSed_
        schedule );
        db.modify( gpo, [&]( auto& gp ) {
            gp.proPoSed_schedule_block_num = optional<block_num_
                type>();
            gp.proPoSed_schedule.clear();
        });
    }

    try {
        auto onbtrx = std::make_shared<transaction_metadata>(
            get_on_block_transaction() );
        auto reset_in_trx_requiring_checks = fc::make_scoped_
            exit([old_value=in_trx_requiring_checks,this](){
                in_trx_requiring_checks = old_value;
            });

        in_trx_requiring_checks = true;
        push_transaction( onbtrx, fc::time_point::maximum(),
            true, self.get_global_properties().configuration.min_
            transaction_cpu_usage );
    } catch( const boost::interprocess::bad_alloc& e ) {
      elog( "on block transaction failed due to a bad allocation" );
      throw;
    } catch( const fc::exception& e ) {
      wlog( "on block transaction failed, but shouldn't impact
          block generation, system contract needs update" );
      edump((e.to_detail_string()));
    } catch( ... ) {
    }

    clear_expired_input_transactions();
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

```
                update_producers_authority();
            }

            guard_pending.cancel();
        } // start_block
```

start_block() 函数首先获取当前区块生产者；接着判断本地节点是否具有最新区块，同步是否完成；接着判断是否具有对应的私钥；正式出块，调用 controller 中的 start_block 函数，生产区块。

3.3　网络层代码实现

P2P 网络是区块链运行的基础模块，在 EOS 中，主要就是 net_plugin、http_plugin、net_pai_plugin。首先看一下网络插件生成时的代码：

```
net_plugin::net_plugin()
    :my( new net_plugin_impl ) {
    my_impl = my.get();
}
```

可以看到，它生成了一个 net_plugin_impl 实例，真正的与网络操作相关的代码都在这个类中。Main 函数中的初始化如下：

```
void net_plugin::plugin_initialize( const variables_map& options ) {
        ilog("Initialize net plugin");
        try {
    ......
//使用boost的resolver来处理与网络相关的数据格式的转换
my->resolver=std::make_shared<tcp::resolver>( std::ref( app().get_io_
service()));
//根据options设置来设置相关配置
if( options.count( "p2p-listen-endpoint" )) {
            my->p2p_address = options.at( "p2p-listen-endpoint" ).as<string>();
            auto host = my->p2p_address.substr( 0, my->p2p_address.find( ':' ));
            auto port = my->p2p_address.substr( host.size() + 1, my->p2p_
                address.size());
            idump((host)( port ));
            tcp::resolver::query query( tcp::v4(), host.c_str(), port.c_str());
//得到监听地址
            my->listen_endpoint = *my->resolver->resolve( query );
//重置boost socket网络接收器
```

非卖品，仅供非商业用途或交流学习使用


```
            my->acceptor.reset( new tcp::acceptor( app().get_io_service()));
        }
        if( options.count( "p2p-server-address" )) {
            my->p2p_address = options.at( "p2p-server-address" ).as<string>();
        } else {
            if( my->listen_endpoint.address().to_v4() == address_v4::any()) {
                boost::system::error_code ec;
                auto host = host_name( ec );
                if( ec.value() != boost::system::errc::success ) {

                    FC_THROW_EXCEPTION( fc::invalid_arg_exception,"Unable to
                        retrieve host_name. ${msg}", ("msg", ec.message()));

                }
                auto port = my->p2p_address.substr( my->p2p_address.find( ':'
                    ), my->p2p_address.size());
                my->p2p_address = host + port;
            }
        }

        if( options.count( "p2p-peer-address" )) {
            my->supplied_peers = options.at( "p2p-peer-address" ).as<vector<string> >();
        }
        if( options.count( "agent-name" )) {
            my->user_agent_name = options.at( "agent-name" ).as<string>();
        }
    //处理连接设置
        if( options.count( "allowed-connection" )) {
            const std::vector<std::string> allowed_remotes = options["allowed-
                connection"].as<std::vector<std::string>>();
            for( const std::string& allowed_remote : allowed_remotes ) {
                if( allowed_remote == "any" )
                    my->allowed_connections |= net_plugin_impl::Any;
                else if( allowed_remote == "producers" )
                    my->allowed_connections |= net_plugin_impl::Producers;
                else if( allowed_remote == "specified" )
                    my->allowed_connections |= net_plugin_impl::Specified;
                else if( allowed_remote == "none" )
                    my->allowed_connections = net_plugin_impl::None;
            }
        }

        ......
    //查找依赖的链插件
```

非卖品，仅供非商业用途或交流学习使用


```
        my->chain_plug = app().find_plugin<chain_plugin>();
        my->chain_id = app().get_plugin<chain_plugin>().get_chain_id();
        fc::rand_pseudo_bytes( my->node_id.data(), my->node_id.data_size());
        ilog( "my node_id is ${id}", ("id", my->node_id));
//重置心跳定时器
        my->keepalive_timer.reset( new boost::asio::steady_timer( app().get_
            io_service()));
        my->ticker();
    } FC_LOG_AND_RETHROW()
}
```

紧接着我们来看以下启动代码：

```
void net_plugin::plugin_startup() {
    if( my->acceptor ) {
//常见的网络服务操作，打开监听服务，设置选项，绑定地址，启动监听
        my->acceptor->open(my->listen_endpoint.protocol());
        my->acceptor->set_option(tcp::acceptor::reuse_address(true));
        my->acceptor->bind(my->listen_endpoint);
        my->acceptor->listen();
        ilog("starting listener, max clients is ${mc}",("mc",my->max_client_
            count));
        my->start_listen_loop();//循环接收连接
    }
//启动连接和交易到期的监视（一个自循环）
    my->start_monitors();
    for( auto seed_node : my->supplied_peers ) {
        connect( seed_node );//连接种子节点，接入P2P网络
    }
}
```

通过 start_listen_loop() 函数来实现循环监听，其主要就是引入了 BOOST 的库，而读取消息与数据的交互用到了 start_read_message() 函数，这里会通过异步的方式来读取真正的数据。

通过上面的函数就完成了服务的监听与接收，接下来看怎么实现主动连接，这里会利用 connect() 来实现，代码如下：

```
void net_plugin_impl::connect( connection_ptr c ) {
    if( c->no_retry != go_away_reason::no_reason) {
        fc_dlog( logger, "Skipping connect due to go_away reason ${r}",("r",
            reason_str( c->no_retry )));
        return;
```

非卖品，仅供非商业用途或交流学习使用


```
    }
    auto colon = c->peer_addr.find(':');

    if (colon == std::string::nPoS || colon == 0) {
        elog ("Invalid peer address. must be \"host:port\": ${p}", ("p",c-
            >peer_addr));
        for ( auto itr : connections ) {
            if((*itr).peer_addr == c->peer_addr) {
                (*itr).reset();
                close(itr);
                connections.erase(itr);
                break;
            }
        }
        return;
    }

    auto host = c->peer_addr.substr( 0, colon );
    auto port = c->peer_addr.substr( colon + 1);
    idump((host)(port));
    tcp::resolver::query query( tcp::v4(), host.c_str(), port.c_str() );
    connection_wptr weak_conn = c;
    // 注意：还需要添加对IPv6的支持
    resolver->async_resolve( query,[weak_conn, this]( const boost::system::
        error_code& err,tcp::resolver::iterator endpoint_itr ){
            auto c = weak_conn.lock();
            if (!c) return;
            if( !err ) {
                connect( c, endpoint_itr );
            } else {
                elog( "Unable to resolve ${peer_addr}: ${error}",
                    ( "peer_addr", c->peer_name() )("error", err.message() ) );
            }
        });
}

void net_plugin_impl::connect( connection_ptr c, tcp::resolver::iterator
    endpoint_itr ) {
    if( c->no_retry != go_away_reason::no_reason) {
        string rsn = reason_str(c->no_retry);
        return;
    }
    auto current_endpoint = *endpoint_itr;
```

```
    ++endpoint_itr;
    c->connecting = true;
    connection_wptr weak_conn = c;
    c->socket->async_connect( current_endpoint, [weak_conn, endpoint_itr,
        this] ( const boost::system::error_code& err ) {
        auto c = weak_conn.lock();
        if (!c) return;
        if( !err && c->socket->is_open() ) {
            if (start_session( c )) {
                c->send_handshake ();
            }
        } else {
            if( endpoint_itr != tcp::resolver::iterator() ) {
                close(c);
                connect( c, endpoint_itr );
            }
            else {
                elog( "connection failed to ${peer}: ${error}",
                    ( "peer", c->peer_name())("error",err.message()));
                c->connecting = false;
                my_impl->close(c);
            }
        }
    } );
}
```

最后要进行的是数据同步。前面已经简单介绍了 start_read_message，但没对其内部实现细节进行分析。

通过前面的代码，我们已经启动网络，并发现了节点，所以这里开始运行 P2P。首先是同步数据，这和之前介绍的比特币数据同步过程类似，也有一个中心的消息处理系统。数据同步的实现代码如下：

```
bool connection::process_next_message(net_plugin_impl& impl, uint32_t message_
    length) {
    try {
        // 如果是signed_block，则保存缓存的原始消息
        // 这必须在我们解压缩消息之前完成
        // 这代码复制于fc::io::unpack(..., unsigned_int)
        auto index = pending_message_buffer.read_index();
        uint64_t which = 0; char b = 0; uint8_t by = 0;
        do {
            pending_message_buffer.peek(&b, 1, index);
```

```
            which |= uint32_t(uint8_t(b) & 0x7f) << by;
            by += 7;
        } while( uint8_t(b) & 0x80 && by < 32);

        if (which == uint64_t(net_message::tag<signed_block>::value)) {
            blk_buffer.resize(message_length);
            auto index = pending_message_buffer.read_index();
            pending_message_buffer.peek(blk_buffer.data(), message_length, index);
        }
        auto ds = pending_message_buffer.create_datastream();
        net_message msg;
        fc::raw::unpack(ds, msg);
        msgHandler m(impl, shared_from_this() );//impl是net_plugin_impl
        msg.visit(m);//注意这里最终调用一个仿函数，该函数位于static_variant.hpp中

    } catch(  const fc::exception& e ) {
        edump((e.to_detail_string() ));
        impl.close( shared_from_this() );
        return false;
    }
    return true;
}
```

3.4 区块打包与广播的代码实现

我们都知道区块生成之后是需要进行打包与广播的，否则会被视为无效的。区块生成之后的 pending 会发送到 push_transactions 中。以下是 push_transaction 的实现代码：

```
transaction_trace_ptr push_transaction( const transaction_metadata_ptr& trx,
fc::time_point deadline,bool implicit,uint32_t billed_cpu_time_us)
    {
        EOS_ASSERT(deadline != fc::time_point(), transaction_exception,
            "deadline cannot be uninitialized");
        transaction_trace_ptr trace;
        try {
            transaction_context trx_context(self, trx->trx, trx->id);
            if ((bool)subjective_cpu_leeway && pending->_block_status ==
                controller::block_status::incomplete) {
                trx_context.leeway = *subjective_cpu_leeway;
            }
```

```
            trx_context.deadline = deadline;
            trx_context.billed_cpu_time_us = billed_cpu_time_us;
            trace = trx_context.trace;
            try {
                if( implicit ) {
                    trx_context.init_for_implicit_trx();
                    trx_context.can_subjectively_fail = false;
                } else {
                    trx_context.init_for_input_trx( trx->packed_trx.get_unprunable_
                        size(),trx->packed_trx.get_prunable_size(),trx->trx.
                        signatures.size());
                }

                if(trx_context.can_subjectively_fail  && pending->_block_
                    status == controller::block_status::incomplete ) {
                    check_actor_list( trx_context.bill_to_accounts );
                    // 假设bill_to_accounts是授权交易的一组参与者            }


                trx_context.delay = fc::seconds(trx->trx.delay_sec);

                if( !self.skip_auth_check() && !implicit ) {
                    authorization.check_authorization(
                        trx->trx.actions,
                        trx->recover_keys( chain_id ),
                        {},
                        trx_context.delay,
                        [](){}
/*std::bind(&transaction_context::add_cpu_usage_and_check_time, &trx_context
    ,std::placeholders::_1)*/,
                        false
                    );
                }
                trx_context.exec();
                trx_context.finalize(); //如果成功，则自动更新交易发起者中的net和
                                        CPU使用情况
                auto restore = make_block_restore_point();

                if (!implicit) {
                    transaction_receipt::status_enum s = (trx_context.delay
                        == fc::seconds(0))? transaction_receipt::executed:
                        transaction_receipt::delayed;
                    trace->receipt = push_receipt(trx->packed_trx, s, trx_
                        context.billed_cpu_time_us, trace->net_usage);
```

```
                pending->_pending_block_state->trxs.emplace_back(trx);
            } else {
                transaction_receipt_header r;
                r.status = transaction_receipt::executed;
                r.cpu_usage_us = trx_context.billed_cpu_time_us;
                r.net_usage_words = trace->net_usage / 8;
                trace->receipt = r;
            }

            fc::move_append(pending->_actions, move(trx_context.executed));

            //若交易被接收，则向网络再次发送这笔交易，且整个过程只能执行一次
            if (!trx->accepted) {
                emit( self.accepted_transaction, trx);
                trx->accepted = true;
            }

            emit(self.applied_transaction, trace);

            if ( read_mode != db_read_mode::SPECULATIVE && pending->_
                block_status == controller::block_status::incomplete ) {
                //这可能会在析构函数中自动发生
                trx_context.undo();
            } else {
                restore.cancel();
                trx_context.squash();
            }

            if (!implicit) {
                unapplied_transactions.erase( trx->signed_id );
            }
            return trace;
        } catch (const fc::exception& e) {
            trace->except = e;
            trace->except_ptr = std::current_exception();
        }

        if (!failure_is_subjective(*trace->except)) {
            unapplied_transactions.erase( trx->signed_id );
        }

        return trace;
    } FC_CAPTURE_AND_RETHROW((trace))
} /// push_transaction
```

在 start_block 中产生的区块是未经多节点确认过的区块，因此这里传入的 implicit 为 true，即这个块或者说这次交易是未确认的状态，此处我们使用 init_for_implicit_trx 对该区块进行初始化。而后，将本次交易的回执信息，如是否执行成功、CPU 的使用情况、net 的使用情况等写入本次交易的回执 trace->receipt 中。

这里需要注意区分 trace、trx、trx_context 之间的不同与联系。trx 包含了本次区块产生的交易信息，trx_context 则是将 trx 的信息写入 trx_context 类中以便接下来使用，而 trace 为 trx_context 中的一个变量类型 action_trace 的值。接下来我们介绍这三者的区别。

本次交易的回执信息填充结束之后，调用 fc::move_append 对 trx_context 使用 move 的方式转化为右值引用，即 move 到区块的 action 中去。move_append 中同样使用了 move，在目标容器为空的情况下将 trx_context 中的内容全部放进去。当目标容器不为空时，会从目标容器末端开始循环插入 trx_context 中的信息，这里所说的目标容器就是 pending->_action，就将交易打包到区块的 _action 中去，这个 _action 为包含有交易回执信息的区块信息。以上操作完成了区块的生产和区块打包的过程。以下是 move_append 的实现代码：

```
fc::move_append(pending->_actions, move(trx_context.executed));

    if (!trx->accepted) {
        emit( self.accepted_transaction, trx);
        trx->accepted = true;
    }
```

区块已经打包完毕后，我们需要看看是怎么把信息广播出去。这里使用了 emit 将 trx 区块内容信息或者将 trace 区块跟踪信息广播出去。emit 的具体实现如下所示：

```
void emit( const Signal& s, Arg&& a ) {
    try {
        s(std::forward<Arg>(a));
    } catch (boost::interprocess::bad_alloc& e) {
        wlog( "bad alloc" );
        throw e;
    } catch ( controller_emit_signal_exception& e ) {
        wlog( "${details}", ("details", e.to_detail_string()) );
        throw e;
    } catch ( fc::exception& e ) {
        wlog( "${details}", ("details", e.to_detail_string()) );
```

仅供非商业用途或交流学习使用

```
        } catch ( ... ) {
            wlog( "signal handler threw exception" );
        }
    }
```

在这里保持了参数的左值与右值的特征，将参数传递给函数模板中调用的另外一个函数。这里 trx 和 trace 均为左值，因其可以赋值且可取址，而后通过完美转发将参数传递给了 Signal。这样 Signal 中便存在着可以使用的右值，如 emit（self.accepted_transaction, trx）和 emit（self.applied_transaction, trace）。熟悉信号槽的人看到 emit 不免会想，这是不是就是信号槽机制？

没错，这正是 boost 中的 signal-slot 的机制。信号会在某个特定情况或动作下被触发，槽是等同于接收并处理信号的函数。做过 qt 开发的人对信号槽机制并不会陌生，拿最简单的 on_pushButton_clicked() 函数来讲，当某一个特定事件发生时（clicked），一个信号被发送（emit），与信号相关联（connect）的槽（slot）则会响应信号并完成相应的处理。而在 boost 中也存在类似的机制，我们结合 EOS 源码中关于区块广播来分析下信号槽的实现。通过跟踪可以发现与 Signal 对应的 slot 均存在于 net_plugin 中。

和大多数信号槽机制一样，在 net_plugin 启动的时候，首先确定信号和槽之间的关系。获取当前链上的绝大多数信息，而后使用 connect 的方式绑定信号，在区块广播出去的过程中并不存在 confirm，因此通过代码跟踪或者日志打印。一个区块产生、打包、广播出去的过程中只包含了 accepted_transaction、applied_transaction、irreversible_block、accepted_block_header、accepted_block，需要注意的是，这里的五个过程是有先后顺序的。区块的广播机制大致如此。

3.5 系统合约层代码实现

什么是合约？在现实世界中，简单来说就是参加一个活动或者进行一次行动，参加的人共同遵守的一个约定或者协议。合约可以是受法律保护的正式合同，也可以是一些简单的游戏规则或者口头协定。典型的活动可以是诸如资金转移（在金融合约的情况下）或游戏动作（在游戏合约的情况下）。而这里所叙述的 EOSIO 系统合约就是在 EOSIO 智能合约的区块链上注册，并在 EOSIO 节点上执行的软件，它实现了一个“契约”的关系，其动作请求需要被存储在区块链上。

仅供非商业用途或交流学习使用

智能合约就是用来定义接口和实现接口的代码，这些接口可以用来实现一些动作、定义一些参数、构造一些数据结构，代码被编译成一个规范的字节码格式，以便节点可以检索和执行。而合约执行的交易记录都会被存储在区块链上，例如货币的交易、动作的移动等。

在 EOSIO 项目中，有 5 个基础的智能合约，分别是 eosio.system、eosio.Token、eosio.msig、eosio.bios 与 eosio.sudo，我们会就其中三个进行分析。

3.5.1　eosio.system 合约代码分析

EOS 的网络带宽和计算资源是通过抵押代币的方式获得的。如果不需要相应资源了，可以解除抵押收回原先的代币。投票的权重和抵押代币所获得的带宽和计算资源相关，抵押越多，投票的权重越大。同时投票的权重会随着时间的推移减少，这个设定鼓励用户持续更新自己的投票。而存储资源则是通过购买得到的，但是并不会消耗，当不再需要存储资源的时候可以出售。但是存储资源相对于 EOS 代币的价格会随着市场对于存储资源的需求而变动。而这些功能都是在 eosio.system 智能合约中实现的。

通过 eosio.system 智能合约可以进行很多系统级别的操作，比如：用户抵押 Token，从而获得投票权投票给区块生成者（block producer），这就是前面所说的 21 个超级节点机制，通过抵押 Token 还可获得社区提案（worker proPoSal）的权利；也可以设置代理，把自己掌握的投票权移交给其他用户；还可以抵押 Token，从而获得相应的网络带宽、存储空间、运算能力。

eosio.system 合约的主要实现代码在 eosio.system.hpp、eosio.system.cpp 两个文件中。eosio.system.hpp 中定义了合约类 eosiosystem::system_contract，和一些结构体，如 eosio_global_state（全局状态）、producer_info（生产者信息）、voter_info（投票人信息）。它们的源代码如下所示：

```
struct eosio_global_state : eosio::blockchain_parameters {
        uint64_t free_ram()const { return max_ram_size - total_ram_bytes_reserved; }

        uint64_t             max_ram_size = 64ll*1024 * 1024 * 1024;
        uint64_t             total_ram_bytes_reserved = 0;
        int64_t              total_ram_stake = 0;

        block_timestamp      last_producer_schedule_update;
        uint64_t             last_pervote_bucket_fill = 0;
```

仅供非商业用途或交流学习使用

```
        int64_t              pervote_bucket = 0;
        int64_t              perblock_bucket = 0;
        uint32_t             total_unpaid_blocks = 0; ///未领取奖励的区块数量
        int64_t              total_activated_stake = 0;
        uint64_t             thresh_activated_stake_time = 0;
        uint16_t             last_producer_schedule_size = 0;
        double               total_producer_vote_weight = 0; ///总投票数
        block_timestamp      last_name_close;

        //显式序列化宏不是必需的，这里仅用于改善编译时间
        EOSLIB_SERIALIZE_DERIVED( eosio_global_state, eosio::blockchain_parameters,
            (max_ram_size)(total_ram_bytes_reserved)(total_ram_stake)
(last_producer_schedule_update)(last_pervote_bucket_fill)

(pervote_bucket)(perblock_bucket)(total_unpaid_blocks)(total_activated_stake)
    (thresh_activated_stake_time)

(last_producer_schedule_size)(total_producer_vote_weight)(last_name_close) )
        };

struct producer_info {
        account_name          owner;
        double                total_votes = 0;
        eosio::public_key     producer_key; ///区块生产者公钥
        bool                  is_active = true;
        std::string           url;
        uint32_t              unpaid_blocks = 0;
        uint64_t              last_claim_time = 0;
        uint16_t              location = 0;
        uint64_t primary_key()const { return owner;}
        double   by_votes()const    { return is_active ? -total_votes : total_votes;}
        bool      active()const     { return is_active;}
        void     deactivate()       { producer_key = public_key(); is_active = false;}

        // 显式序列化宏不是必须的，这里仅用于改善编译时间
        EOSLIB_SERIALIZE(producer_info,(owner)(total_votes)(producer_key)(is_active)(url)
            (unpaid_blocks)(last_claim_time)(location) )
        };

    struct voter_info {
        account_name                owner = 0; /// 选民
        account_name                proxy = 0; /// 选民设定的代理人
        std::vector<account_name>   producers; /// 被选出块节点账户生产者
        int64_t                     staked = 0;
        double                      last_vote_weight = 0; /// 最后一次投票更新投票
                                                          权重
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
        double                          proxied_vote_weight= 0; ///若为代理人，则表示对
                                                                该投票拥有的权重
        bool                            is_proxy = 0; /// 选民是否是其他人的代理人
        uint32_t                        reserved1 = 0;
        time                            reserved2 = 0;
        eosio::asset                    reserved3;
        uint64_t primary_key()const { return owner; }
        // 显式序列化宏不是必须的，这里仅用于改善编译时间
        EOSLIB_SERIALIZE( voter_info, (owner)(proxy)(producers)(staked)(last_
            vote_weight)(proxied_vote_weight)(is_proxy)(reserved1)(reserved2)
            (reserved3) )
    };
```

eosio.system 合约中包含很多 action，其通过 eosio.system.cpp 中的 EOSIO_ABI 宏定义的代码如下所示。从代码中可以看出，action 的具体代码实现分散到了 nativ.cpp、delegate_bandwith.cpp、voting.cpp、producer_pay.cpp 中。

```
EOSIO_ABI( eosiosystem::system_contract,
    // native.hpp (newaccount definition is actually in eosio.system.
        cpp)(newaccount)(updateauth)(deleteauth)(linkauth)(unlinkauth)
        (canceldelay)(onerror)
    // eosio.system.cpp
    (setram)(setparams)(setpriv)(rmvproducer)(bidname)
    // delegate_bandwidth.cpp
    (buyrambytes)(buyram)(sellram)(delegatebw)(undelegatebw)(refund)
    // voting.cpp
    (regproducer)(unregprod)(voteproducer)(regproxy)
    // producer_pay.cpp
    (onblock)(claimrewards)
)
```

native.cpp 的作用是对创建的新账户进行可行性验证。在创建新用户时调用此类，用来约束新账户的命名规则。在账户创建的时候不可以包含特殊的字符，同时规定账户的名称在 12 个字符以内，防止账户重复。新账户必须包含最少数量的代币（在系统参数中设置），因此，此方法将从接收器 newaccount 中执行内联 buyram 操作，购买 ram 的费用相当于当前新账户创建费用的金额。代码如下所示：

```
class native : public eosio::contract {
    public:
        using eosio::contract::contract;
        //  等于当前新账户创建费用的金额
        void newaccount( account_name creator,account_name newact
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
        void updateauth( /*account_name     account,
            permission_name  permission,
            permission_name  parent,
            const authority& data*/ ) {}
        void deleteauth( /*account_name account, permission_name permission*/ )
            {}
        void linkauth( /*account_name    account,
            account_name    code,
            action_name     type,
            permission_name requirement*/ ) {}
        void unlinkauth( /*account_name account,
            account_name code,
            action_name  type*/ ) {}
        void canceldelay( /*permission_level canceling_auth, transaction_id_
            type trx_id*/ ) {}
        void onerror( /*const bytes&*/ ) {}
    };
```

delegate_bandwith.cpp 主要是与资源有关的东西。Delegatebw 函数是用来通过抵押 Token 来获取网络和 CPU 资源，其中 account_name from 是 Token 的抵押者；account_name receiver 是 Token 的接收者（表示抵押获取的资源作用在这个账号上）；asset stake_net_quantity 与 asset stake_cpu_quantity 分别表示用来获取网络与 CPU 资源所应抵押代币的数量；transfer 指的是交易进行的状态。Ubdelegatebw 的作用是用来取消抵押，释放网络和 CPU 资源，收回代币，对应的参数与上面的相同，但是一旦 account_name from 取消抵押后，其会失去投票权，解除抵押之后，资源会马上释放，同时投票权重也相应消失。

取消抵押之后，代币需要等待一定时间才能回到账号里；Buyram 是用来购买指定价值内存时调用的，account_name payer 是购买者，account_name receiver 是内存接收者，quant 是用来购买内存所用的代币；Buyrambytes 购买指定大小的内存，支付的 EOS 会以当前市场价格计算；Sellram 函数功能是出售不需要的存储资源；Refund 函数在通过 undelegatebw 函数解除抵押后调用，作用是把抵押的代币（Token）退回账户，退回账户大约有三天的等待期。上述函数的实现代码如下所示：

```
void system_contract::delegatebw( account_name from, account_name receiver,
            asset stake_net_quantity,
            asset stake_cpu_quantity, bool transfer )
    {
        eosio_assert( stake_cpu_quantity >= asset(0), "must stake a PoSitive
            amount" );
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
        eosio_assert( stake_net_quantity >= asset(0), "must stake a PoSitive
            amount" );
        eosio_assert( stake_net_quantity + stake_cpu_quantity > asset(0),
            "must stake a PoSitive amount" );
        eosio_assert( !transfer || from != receiver, "cannot use transfer
            flag if delegating to self" );

        changebw( from, receiver, stake_net_quantity, stake_cpu_quantity,
            transfer);
    } // delegatebw

void system_contract::undelegatebw( account_name from, account_name receiver,
    asset unstake_net_quantity, asset unstake_cpu_quantity )
    {
        eosio_assert( asset() <= unstake_cpu_quantity, "must unstake a
            PoSitive amount" );
        eosio_assert( asset() <= unstake_net_quantity, "must unstake a
           PoSitive amount" );
        eosio_assert( asset() < unstake_cpu_quantity + unstake_net_quantity,
            "must unstake a PoSitive amount" );
        eosio_assert( _gstate.total_activated_stake >= min_activated_
            stake,"cannot undelegate bandwidth until the chain is activated
            (at least 15% of all Tokens participate in voting)" );

        changebw( from, receiver, -unstake_net_quantity, -unstake_cpu_
            quantity, false);
    } // undelegatebw

void system_contract::buyram( account_name payer, account_name receiver,
    asset quant )
    {
        require_auth( payer );
        eosio_assert( quant.amount > 0, "must purchase a PoSitive amount" );
        auto fee = quant;
        fee.amount = ( fee.amount + 199 ) / 200; /// .5% fee (round up)
        // fee.amount不能为0，只有当quant.amount为0且上述断言不允许时才可以
        // If quant.amount == 1, then fee.amount == 1,
        // otherwise if quant.amount > 1, then 0 < fee.amount < quant.amount.
        auto quant_after_fee = quant;
        quant_after_fee.amount -= fee.amount;
        ......
        set_resource_limits( res_itr->owner, res_itr->ram_bytes, res_itr-
            >net_weight.amount, res_itr->cpu_weight.amount );
    }
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
void system_contract::buyrambytes( account_name payer, account_name receiver,
    uint32_t bytes ) {
        auto itr = _rammarket.find(S(4,RAMCORE));
        auto tmp = *itr;
        auto eosout = tmp.convert( asset(bytes,S(0,RAM)), CORE_SYMBOL );

        buyram( payer, receiver, eosout );
    }

void system_contract::sellram( account_name account, int64_t bytes ) {
        require_auth( account );
        eosio_assert( bytes > 0, "cannot sell negative byte" );
        user_resources_table  userres( _self, account );
        auto res_itr = userres.find( account );
        eosio_assert( res_itr != userres.end(), "no resource row" );
        eosio_assert( res_itr->ram_bytes >= bytes, "insufficient quota" );
.......
        if( fee > 0 ) {
            INLINE_ACTION_SENDER(eosio::Token, transfer)( N(eosio.Token),
                {account,N(active)},
            { account, N(eosio.ramfee), asset(fee), std::string("sell ram fee") } );
        }
    }
```

Voting.cpp 主要包含与投票相关的 action。Regproducer 用来注册生产者，也就是 EOSIO 中的超级节点候选人，其会为生产者创建或更新 producer_info 对象，这里包括账户名、账户公钥、候选节点的网站地址、候选节点的机房地理位置等信息；Unregprod 用来取消成为候选人账户的注册；voteproducer 在对账户进行投票时使用，在这里是有两种投票模式，即代理模式和直接投票模式。代理模式是将投票权重委托给一个代理人，让他帮忙投票。

直接投票模式就是直接投票给你信任的超级节点。一个账户可以投给一个或者多个生产者，最多可以为 30 个。regproxy 函数的作用是注册成为投票代理人，代理人接受其他用户的委托。需要注意的是，已经使用了代理的投票者，不能注册成为投票代理人。上述函数代码如下所示：

```
void system_contract::regproducer( const account_name producer, const
    eosio::public_key& producer_key, const std::string& url, uint16_t
    location ) {
        eosio_assert( url.size() < 512, "url too long" );
        eosio_assert( producer_key != eosio::public_key(), "public key
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            should not be the default value" );
        require_auth( producer );

        auto prod = _producers.find( producer );

        if ( prod != _producers.end() ) {
            _producers.modify( prod, producer, [&]( producer_info& info ){
                info.producer_key = producer_key;
                info.is_active    = true;
                info.url          = url;
                info.location     = location;
            });
        } else {
            _producers.emplace( producer, [&]( producer_info& info ){
                info.owner          = producer;
                info.total_votes    = 0;
                info.producer_key   = producer_key;
                info.is_active      = true;
                info.url            = url;
                info.location       = location;
            });
        }
    }

    void system_contract::unregprod( const account_name producer ) {
        require_auth( producer );

        const auto& prod = _producers.get( producer, "producer not found" );

        _producers.modify( prod, 0, [&]( producer_info& info ){
            info.deactivate();
        });
    }
void system_contract::voteproducer( const account_name voter_name, const
    account_name proxy, const std::vector<account_name>& producers ) {
        require_auth( voter_name );
        update_votes( voter_name, proxy, producers, true );
    }

void system_contract::regproxy( const account_name proxy, bool isproxy ) {
        require_auth( proxy );

        auto pitr = _voters.find(proxy);
        if ( pitr != _voters.end() ) {
            eosio_assert( isproxy != pitr->is_proxy, "action has no effect" );
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            eosio_assert( !isproxy || !pitr->proxy, "account that uses a
                proxy is not allowed to become a proxy" );
            _voters.modify( pitr, 0, [&]( auto& p ) {
                p.is_proxy = isproxy;
              });
            propagate_weight_change( *pitr );
        } else {
            _voters.emplace( proxy, [&]( auto& p ) {
                p.owner  = proxy;
                p.is_proxy = isproxy;
            });
        }
    }
```

其中，voting.cpp 中包含一个比较重要的 action。update_elected_producers 就是用这个函数来选出 EOSIO 项目中的 21 个超级节点的，这里是根据投票数由高到低选出 21 个区块生产者，并在一定的时间间隔内更新这些数据，前面我们已经对其进行分析了，在这里只进行局部展开。

producer_pay.cpp 是用来计算生产者节点收益的相关代码。其中 onblock 用来计算一些遗漏的区块，更新指定生产者的区块信息。而 claimrewards 用来计算生产者生产的区块数量，计算每个块的收益，计算根据生产者的生产时间，计算每秒收益，计算生产者从每次投票中获得的收益等，综合以上的收益来计算出生产者（超级节点）的总收益。

```
void system_contract::onblock( block_timestamp timestamp, account_name producer )
{
    using namespace eosio;
    require_auth(N(eosio));
    ......
    if( highest != idx.end() &&
        highest->high_bid > 0 &&
        highest->last_bid_time < (current_time() - useconds_per_day) &&
        _gstate.thresh_activated_stake_time > 0 &&
        (current_time() - _gstate.thresh_activated_stake_time) > 14 *
            useconds_per_day ) {
            _gstate.last_name_close = timestamp;
            idx.modify( highest, 0, [&]( auto& b ){
                b.high_bid = -b.high_bid;
        });
    }
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

```
}
using namespace eosio;
void system_contract::claimrewards( const account_name& owner ) {
    require_auth(owner);
    ......
    if( producer_per_vote_pay > 0 ) {
        INLINE_ACTION_SENDER(eosio::Token, transfer)( N(eosio.Token),
            {N(eosio.vpay),N(active)},
            {N(eosio.vpay), owner, asset(producer_per_vote_pay),
               std::string ("producer vote pay") } );
    }
}
```

在这里我们就可以通过此合约来测试网络成为区块链生产者的收益。通过 cleos system 子命令来测试相应功能，测试的主要步骤如下：首先注册账号，同时启动节点与主网同步区块；其次注册账号成为候选人参加竞选；再次抵押自己的代币来获取资源与投票的权重，目的是增加自己的话语权用来投票给自己；最后当自己获得足够多的票数时，等待一定的时间之后就可以成为出块人了，也就是 EOSIO 中的超级节点。

3.5.2　eosio.Token 合约代码分析

在 EOS 目录自带的合约中，有一个 eosio.Token 智能合约，这个智能合约可以用于生产代币（Token），同时可以把合约上的 Token 分发给不同的账号去管理。这些 Token 可用于转账操作。eosio.Token 中主要的代码逻辑是在 eosio.Token.cpp 中实现的，主要的函数有：

（1）create 函数：功能是创建代币，使用正确的参数调用该函数。可以按照最大需要去创建定义的代币，创建这个 Token 时发行人将有权发出问题并执行其他操作，例如冻结、召回和列入所有者白名单。

```
void Token::create( account_name issuer,asset maximum_supply )
{
//获取授权，如果没有授权，action调用会中止，事务会回滚
    require_auth( _self );
    auto sym = maximum_supply.symbol;
    eosio_assert( sym.is_valid(), "invalid symbol name" );
    eosio_assert( maximum_supply.is_valid(), "invalid supply");
    eosio_assert( maximum_supply.amount > 0, "max-supply must be PoSitive");
//设置状态机，授权账户创建这个标志的币种
    stats statstable( _self, sym.name() );
```

非卖品，仅供非商业用途或交流学习使用

```
//检查状态机中是否存在这个标志的币种
    auto existing = statstable.find( sym.name() );
    eosio_assert( existing == statstable.end(), "Token with symbol already
        exists" );
//更新状态机，将资产数量、资产发行者、资产标志一起写入状态机中
    statstable.emplace( _self, [&]( auto& s ){
        s.supply.symbol = maximum_supply.symbol;
        s.max_supply    = maximum_supply;
        s.issuer        = issuer;
    });
}
```

（2）issue 函数：来实现资产的发放。发行者可以向其他账户发放创建的 Token，这样账户就具有此 Token 的使用权与交易权。其中，account_name to 用来接收发放 Token 的用户；asset quantity 表示发放的代币数量；string memo 用于发放代币时添加备注。

```
void Token::issue( account_name to, asset quantity, string memo )
{
    auto sym = quantity.symbol;
    eosio_assert( sym.is_valid(), "invalid symbol name" );
    eosio_assert( memo.size() <= 256, "memo has more than 256 bytes" );
    auto sym_name = sym.name();
//获取状态机信息
    stats statstable( _self, sym_name );
    auto existing = statstable.find( sym_name );
//检查转给标志的币种是否存在，如果不存在，action调用中止，事务回滚
    eosio_assert( existing != statstable.end(), "Token with symbol does not
        exist, create Token before issue" );
    const auto& st = *existing;

//获取授权，如果没有授权，action调用中止，事务回滚
// 检查发行人权限，判断是否达到权限阈值
    require_auth( st.issuer );
// 转移金额是否有效
    eosio_assert( quantity.is_valid(), "invalid quantity" );
// 转移金额必须是正数
    eosio_assert( quantity.amount > 0, "must issue PoSitive quantity" );

    eosio_assert( quantity.symbol == st.supply.symbol, "symbol precision
        mismatch" );
    eosio_assert( quantity.amount <= st.max_supply.amount - st.supply.
        amount, "quantity exceeds available supply");
```

非卖品，仅供非商业用途或交流学习使用

```
//修改状态机信息
    statstable.modify( st, 0, [&]( auto& s ) {
        s.supply += quantity;
    });
//调用add_balance函数，修改状态机信息
    add_balance( st.issuer, quantity, st.issuer );
//通过调用inline transfer函数后，将交易写入区块中，完成这笔交易
    if( to != st.issuer ) {
        SEND_INLINE_ACTION( *this, transfer, {st.issuer,N(active)}, {st.
            issuer, to, quantity, memo} );
    }
}
```

（3）Transfer 函数：用来实现账户之间代币的交易。其中，account_name from 指的是转出代币的账户；account_name to 指的是接收代币的账户，在这里会对这两个账户进行身份的验证，同时两个账户不可以一样，也就是自己不可以转账给自己；asset quantity 指的是进行交易时代币的数量，这里会对转出账户的余额进行验证，如果小于转出的数量，则请求不会被执行的；memo 指的是在进行交易的时候可以添加备注，函数中对备注的大小进行了限制，在 256 个字节以内。详细的函数实现如下：

```
void Token::transfer( account_name from,account_name to,asset quantity,string
    memo ){
    eosio_assert( from != to, "cannot transfer to self" );
//获取授权，如果没有授权，action调用中止，事务回滚
    require_auth( from );
// 检查账户合法性
    eosio_assert( is_account( to ), "to account does not exist");
    auto sym = quantity.symbol.name();
    stats statstable( _self, sym );
    const auto& st = statstable.get( sym );
//验证账户信息
    require_recipient( from );
    require_recipient( to );
// 检查转账的数量和代币正常性
    eosio_assert( quantity.is_valid(), "invalid quantity" );
    eosio_assert( quantity.amount > 0, "must transfer PoSitive quantity" );
    eosio_assert( quantity.symbol == st.supply.symbol, "symbol precision
        mismatch" );
    eosio_assert( memo.size() <= 256, "memo has more than 256 bytes" );

//调用减掉资产的函数，如果不成功，action调用中止，事务回滚
    sub_balance( from, quantity );
```

非卖品，仅供非商业用途或交流学习使用


```
    //调用增加资产的函数，如果不成功，action调用中止，事务回滚
        add_balance( to, quantity, from );
    }
```

（4）sub_balance 函数与 add_balance 函数：辅助函数。从 transfer 函数的代码中可以看出，当进行账户间代币交易的时候会调用这两个函数，通过 sub_balance 函数扣除交易时的代币数量，通过 add_balance 函数把转账时的代币数量添加到对应的账户上。需要注意的是，如果代币接收方此前没有这种代币的任何交易记录，则发送方需要消耗 RAM 以在“to_acnts”表中创建记录并更新余额；如果此前有记录，那么不需再支付 RAM，直接增加余额就行。这个过程相当于对一个新的用户进行初始化时需要消耗 RAM。

```
void Token::add_balance( account_name owner, asset value, account_name ram_payer )
{
    accounts to_acnts( _self, owner );
    auto to = to_acnts.find( value.symbol.name() );
//用来判断接收账户此前是否有创建记录
    if( to == to_acnts.end() ) {
        to_acnts.emplace( ram_payer, [&]( auto& a ){
            a.balance = value;
        });
    } else {
        to_acnts.modify( to, 0, [&]( auto& a ) {
            a.balance += value;
        });
    }
}
```

3.5.3 eosio.msig 合约代码分析

Msig 为 multi-signature 的缩写，是多重签名的意思。该合约主要提供多方对同一笔交易发起异步签名的功能，通过调用该合约，用户可以便捷地对某一提案进行异步批复。在这个合约的文件夹中包含了 abi、cpp 以及 hpp 三个文件。其中 abi 是一个 JSON 格式的文件，用来描述对 cpp 文件中的 action 和二进制程序进行转变的方法，也用来描述数据库状态。此描述可以使开发者和用户通过 JSON 无缝地与合约进行交互。hpp 文件是头文件，包括了 cpp 文件要使用的变量、常量及引用方法。

cpp 文件是源文件，里面编写了具体的合约执行方法。其中具体实现的方法有：

proPoSe、approve、unapprove、cancel、exec 等。其中 proPoSe 函数用于获取账户的提案与记录提案，为了节约资源，提案中的信息会在函数执行的时候自行读取、比对与存储。具体实现如下所示：

```
void multisig::proPoSe() {
//获取action的大小与数据
    constexpr size_t max_stack_buffer_size = 512;
    size_t size = action_data_size();
    char* buffer = (char*)( max_stack_buffer_size < size ? malloc(size) :
        alloca(size) );
    read_action_data( buffer, size );

//获取提案人的信息与提案信息
    account_name proPoSer;
    name proPoSal_name;
    vector<permission_level> requested;
    transaction_header trx_header;
    datastream<const char*> ds( buffer, size );
    ds >> proPoSer >> proPoSal_name >> requested;

    size_t trx_PoS = ds.tellp();
    ds >> trx_header;
//获取提案人权限，并获取最新提案
    require_auth( proPoSer );
    eosio_assert( trx_header.expiration >= eosio::time_point_sec(now()),
        "transaction expired" );
//对最新的提案进行判断，查看是否重复提交
    proPoSals proptable( _self, proPoSer );
    eosio_assert( proptable.find( proPoSal_name ) == proptable.end(), "proPoSal
        with the same name exists" );
//对此提案进行打包，并判断是否具有交易权限
    bytes packed_requested = pack(requested);
    auto res = ::check_transaction_authorization( buffer+trx_PoS, size-trx_
        PoS,(const char*)0, 0,packed_requested.data(),packed_requested.size() );
    eosio_assert( res > 0, "transaction authorization failed" );

    proptable.emplace( proPoSer, [&]( auto& prop ) {
        prop.proPoSal_name       = proPoSal_name;
        prop.packed_transaction  = bytes( buffer+trx_PoS, buffer+size );
    });
//自我同意此提案，并把记录存储到表中
    approvals apptable(  _self, proPoSer );
    apptable.emplace( proPoSer, [&]( auto& a ) {
        a.proPoSal_name       = proPoSal_name;
```

```
            a.requested_approvals = std::move(requested);
        });
    }
```

approve 函数的功能为对某提案进行同意操作，进而修改本地代码。举个例子，某用户创建了对系统合约的提案之后，超级节点需要对提案发起评审，并决定是否同意该提案，当同意该提案的节点超过一定数量时，提案即可生效。这一数量由 threshold 决定。参数 account_name proPoSer 指的是此交易提案人（也就是申请人），name proPoSal_name 指的是提案名，permission_level level 指的是使用的哪个权限来批准这个提案。unapprove 函数对应的参数功能与 approve 函数相同，所以这里只对 approve 函数做详细说明，具体如下：

```
void multisig::approve( account_name proPoSer, name proPoSal_name, permission_
    level level ) {
    require_auth( level );
//获取提案人的信息
    approvals apptable(  _self, proPoSer );
    auto& apps = apptable.get( proPoSal_name, "proPoSal not found" );
//对提案人进行信息的校验
    auto itr = std::find( apps.requested_approvals.begin(), apps.requested_
        approvals.end(), level );
    eosio_assert( itr != apps.requested_approvals.end(), "approval is not on
        the list of requested approvals" );
//如校验成功，则同意修改该提案
    apptable.modify( apps, proPoSer, [&]( auto& a ) {
        a.provided_approvals.push_back( level );
        a.requested_approvals.erase( itr );
    });
}
```

cancel 函数是取消提案，在这里只需要传递三个参数，其中 proPoSer 指的提议者名称，proPoSal_name 指的是待取消的提案名，参数 canceler 指的是发起取消操作的账户名。exec 函数执行取消操作。

总结一下，cancel 函数的逻辑就是，当提案得到足够数量的节点批准之后，会执行取消操作。

3.5.4 eosiolib 依赖库

eosiolib 库位于 eos/contracts 目录下，上面所叙述的 EOS 基础智能合约和其他示

例合约都会用到 eosiolib 库中定义的类和函数。Eosiolib 文件夹中包含有 40 多个文件，每个文件都有不同的作用，下面我们挑选出一些比较重要的文件进行简单的介绍。图 3-25 所示是 EOSlib 所包含的源文件。

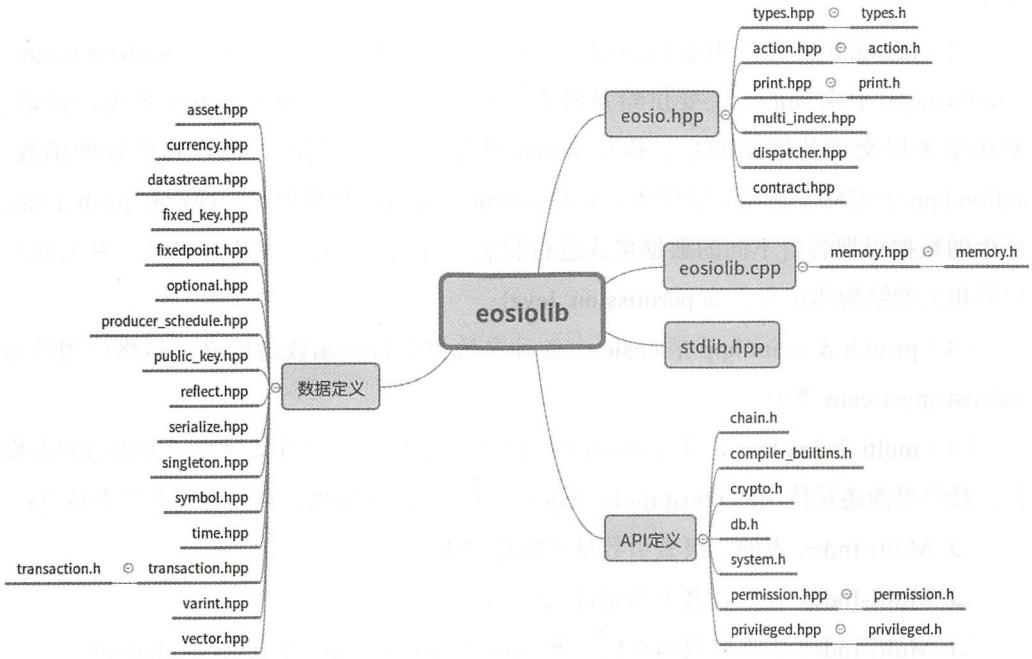


图 3-25　EOSlib 包含的源文件

eosio.hpp 文件类似一个文件引入源，引入了一些必要的其他源文件，即一个 include 的集合。在开发智能合约时，一般都包含这个头文件，示例代码如下：

```
/**
 *  @file
 *  @copyright defined in eos/LICENSE.txt
 */
#pragma once
//引入各种源文件
#include <eosiolib/types.hpp>
#include <eosiolib/action.hpp>
#include <eosiolib/print.hpp>
#include <eosiolib/multi_index.hpp>
#include <eosiolib/dispatcher.hpp>
#include <eosiolib/contract.hpp>
```

对 EOSlib 源文件目录中涉及的目录详述如下：

（1）types.hpp 与 types.p 定义了 EOS 项目中的内建类型和相关的函数。eosiolib 所有使用到的类型都在这里定义，name 标识符与 std::string 类型之间转换函数也在这里定义。

（2）action.h 头文件为接口函数，这些 C API 函数是在 eos/libraries/chain/wasm_interface.cpp 中实现的，主要供给基础合约使用。其主要功能有 action 数据的读取、发送事务提交后的通知消息、执行 action 所需要的权限判断、包含其他辅助函数。action.hpp 中定义了 action 结构体，它是 action 打包后的数据形式，即所有 push action 产生的数据根据各自不同的数据格式进行封装、打包和传送。其中还涉及一些与执行权限相关的结构体定义，如 permission_level。

（3）print.h & print.hpp 是 eosio 中各种参数类型打印函数的定义，最终使用的是 std::ostringstream 类型。

（4）multi_index.hpp 是基于 boost 库的多索引数据表。它重新封装了多索引容器模板，最终里面还是使用的 boost:multi_index，这一点没有变化，它具有以下几个特点：

- Multi-Index 表的二级索引数量不能超过 16 个。
- Multi-Index 表的表名不能超过 12 个字符。
- Multi-Index 表支持双向迭代，即 const_iterator 和 const_reverse_iterator。

很多类型都使用了它来存放，如生产者，投票，单例等。

（5）dispatcher.hpp 用来处理、分发传入的 action。它定义了执行 action 的函数模板 execute_action，使用远程 API 调用 Action 时，会执行这个函数。其中有三个宏：

- EOSIO_API_CALL 用来调用内部的 execute_action 函数，执行一个 Action。
- EOSIO_API 使用 BOOST_PP_SEQ_FOR_EACH 循环调用 EOSIO_API_CALL，即调用多个 Action。
- EOSIO_ABI 里面是一个 apply 函数，后面还会调用 EOSIO_API 这个宏，即它也是用来调用 execute_action 来执行 action 的，在我们自行编写智能合约的时候在类内部最后会加上这个宏。所以当我们加载自己编写的智能合约时，会把编写好的指定的函数功能放到 action 上面去执行。

```
// 定义EOSIO_API_CALL
#define EOSIO_API_CALL( r, OP, elem ) \
```

```
    case ::eosio::string_to_name( BOOST_PP_STRINGIZE(elem) ): \
        eosio::execute_action( &thiscontract, &OP::elem ); \
        break;

// 定义EOSIO_API
#define EOSIO_API( TYPE,  MEMBERS ) \
    BOOST_PP_SEQ_FOR_EACH( EOSIO_API_CALL, TYPE, MEMBERS )

//定义EOSIO_ABI
#define EOSIO_ABI( TYPE, MEMBERS ) \
extern "C" { \
    void apply( uint64_t receiver, uint64_t code, uint64_t action ) { \
        auto self = receiver; \
        if( action == N(onerror)) { \
            /* onerror is only valid if it is for the "eosio" code account
                and authorized by "eosio"'s "active permission */ \
            eosio_assert(code == N(eosio), "onerror action's are only valid
                from the \"eosio\" system account"); \
        } \
        if( code == self || action == N(onerror) ) { \
            TYPE thiscontract( self ); \
            switch( action ) { \
                EOSIO_API( TYPE, MEMBERS ) \
            } \
            /* does not allow destructor of thiscontract to run: eosio_exit(0); */ \
        } \
    } \
} \
```

（6）contract.hpp 定义了所有智能合约的基类，如果自己写合约，需要从这个基类派生，然后使用 eosiocpp 工具将自己的头文件和源文件生成 wasm 文件及其相关的接口文件。编写智能合约需要注意各个文件名需要名称相同，因为加载智能合约时填写的参数是根据智能合约文件路径来的。

（7）eosiolib.cpp 提供内存管理、数据流、区块链参数设置等功能。其中 memory.hpp 与 memory.h 用于内存管理。在这里，对常见的内存分配方式进行了重新编写（其实只是简单的封装而已）。

（8）stdlib.hpp 是一个 include 集合，引用了 C++ 标准库中的 initializer_list、iterator 和 memory。

（9）chain.h 用来查询区块链内部状态的 API，目前用于获取活跃（Active）的生产者，是从 controller 中拿取的，源码在 eos/libraries/chain/wasm_interface.cpp 中实现。

（10）compiler_builtins.h 用于工具链生成编译器内嵌部件的 API。

（11）crypto.h 定义各种加密算法。对给定的数据进行各种 hash 运算并进行存储，或者对数据进行校验。源码在 eos/libraries/chain/wasm_interface.cpp 中实现。

（12）db.h 用于和区块链数据库交互的 API 接口，db.h 函数都在 eos/libraries/chain/wasm_interface.cpp 中的 database_api 类中定义，部分直接定义，部分采用宏定义，源码实现在 eos/libraries/chain/apply_context.cpp 中。提供了数据库的存储与查询的基本功能。其中使用的数据是 controller 类中定义的数据，在这里只是一个引用。

（13）system.h 定义用于与系统级内部函数进行交互的 API，如系统时间函数和静态断言的定义。

（14）permission.hpp 与 permission.h 用来检测事务相关权限，在 eos/libraries/chain/wasm_interface.cpp 的 permission_api 类中实现，事务提交时需要检验执行者的权限都在这里。具体的校验通过与 controller 定义的数据库比对实现。

（15）privileged.hpp 与 privileged.h 实现系统相关的授权，定义特权用户的各种资源限制、事务相关参数、块相关参数及数据传输时间间隔的各种限制。

（16）asset.hpp 定义账户资产，即用一个 int64_t 表示资金，用 symbol_type 表示单位（目前被定义为 SYS）。

（17）currency.hpp 定义有关货币（Token），所有有关货币的创建、发行、转账、余额查询都在这里实现。

（18）datastream.hpp 定义数据流操作，可以把自定义的类的数据成员放到 vector<char> 中，需要时可以再次读取出来，方便数据的传输。

（19）fixed_key.hpp 用于将固定大小 key 按键按字典顺序排序，使用数据成员 std::array<word_t, num_words()> 实现。

（20）fixedpoint.hpp 是 32/64/128/256 位版本的不动点变量。

（21）optional.hpp 表示可选类型，与 boost:optinal 基本一致。

（22）producer_schedule.hpp 表示生产者（出块者）计划表。

（23）public_key.hpp 公钥相关的数据。

（24）reflect.hpp 是 c++ 实现的反射器，带有大量的宏。

（25）serialize.hpp 实现序列化宏，可以对有关数据进行序列化。

（26）singleton.hpp 用于实现单例模式的封装，也存储在 multi_index 中。

（27）symbol.hpp 用于定义代币符号，规定为 SYS。

（28）time.hpp 用于与时间相关的数据，类似于 boost 中的实现。

（29）transaction.hpp 与 transaction.h 事务相关，主要是对 action 用 vector 封装，另外，对 nsaction 的数据传输也做了各种限制，如预期时间、带宽、CPU 算力、延迟传输等。还有一个 get action 函数的实现，可以获取当前区块相关的数据。

（30）varint.hpp 定义整型，在 datastream 中有应用。

（31）vector.hpp 定义字节数组。

3.6　EOS 关键工具的代码实现

EOS 项目网络中提供了三大工具供开发者开发 EOS 应用时使用分别是可以运行单节点的 nodeos、管理 EOS 钱包部件的 keosd 和用于命令符界面的 cleos。这三个工具是开发智能合约、生产 Token 的必备工具。

nodeos 是运行在服务端的区块链节点组件，是 EOSIO 系统的核心进程，可以通过它运行一个节点。keosd 是 EOSIO 的钱包管理客户端，可以被认为是一个存储公钥和私钥的仓库，同时管理钱包信息。nodeos 与 keosd 之间并不存在必然关联，只有在需要签名时它们才会产生联系，例如为交易签名。有一点要注意，nodeos 已经包含了 keosd 的完整功能，也就是说在 nodeos 上也可以管理钱包。cleos 是用户与 keosd/nodeos 交互的命令行工具，可以说是一个连接用户与 keosd/nodeos 的桥梁。

3.6.1　nodeos 代码实现

nodeos 为 blockchain node 组件，其位于 ./eos/build/programs/nodeos/nodeos。nodeos 配置文件夹位于：~/.local/share/eosio/nodeos/config，里面包括 config.ini 和 genesis.json 两个文件。nodeos 启动后会去配置文件夹查找 config.ini 文件，如果没有找到，则会创建一个默认的 config.ini 文件。nodeos 用户数据文件夹位于：~/.local/share/eosio/nodeos/data，存储 runtime data（包括 shared memory 和 log content）。nodeos 是 EOS 区块链的服务器，程序入口为 main() 函数：

```
int main(int argc, char** argv)
{
```

```
    try {
//设置版本号
        app().set_version(eosio::nodeos::config::version);
        app().register_plugin<history_plugin>();
//设置数据文件保存路径
        auto root = fc::app_path();
        app().set_default_data_dir(root / "eosio/nodeos/data" );
//设置配置文件保存路径
        app().set_default_config_dir(root / "eosio/nodeos/config" );
//完成4个插件的初始化工作，如果初始化失败则程序停止运行
        if(!app().initialize<chain_plugin, http_plugin, net_plugin,
            producer_plugin>(argc, argv))
            return INITIALIZE_FAIL;
//日志的初始化
        initialize_logging();
//记录日志
        ilog("nodeos version ${ver}", ("ver", eosio::utilities::common::itoh
            (static_cast<uint32_t>(app().version()))));
        ilog("eosio root is ${root}", ("root", root.string()));
//启动
        app().startup();
//运行
        app().exec();
    } catch( const extract_genesis_state_exception& e ) {
        return EXTRACTED_GENESIS;
    } catch( const fixed_reversible_db_exception& e ) {
        return FIXED_REVERSIBLE;
    } catch( const node_management_success& e ) {
        return NODE_MANAGEMENT_SUCCESS;
    } catch( const fc::exception& e ) {
        if( e.code() == fc::std_exception_code ) {
            if( e.top_message().find( "database dirty flag set" ) !=
               std::string::nPoS ) {
                elog( "database dirty flag set (likely due to unclean
                    shutdown): replay required" );
                return DATABASE_DIRTY;
            } else if( e.top_message().find( "database metadata dirty flag
                set" ) != std::string::nPoS ) {
                elog( "database metadata dirty flag set (likely due to
                    unclean shutdown): replay required" );
                return DATABASE_DIRTY;
            }
        }
        elog( "${e}", ("e", e.to_detail_string()));
        return OTHER_FAIL;
```

```
    } catch( const boost::interprocess::bad_alloc& e ) {
        elog("bad alloc");
        return BAD_ALLOC;
    } catch( const boost::exception& e ) {
        elog("${e}", ("e",boost::diagnostic_information(e)));
        return OTHER_FAIL;
    } catch( const std::runtime_error& e ) {
        if( std::string(e.what()) == "database dirty flag set" ) {
            elog( "database dirty flag set (likely due to unclean shutdown):
                replay required" );
            return DATABASE_DIRTY;
        } else if( std::string(e.what()) == "database metadata dirty flag set" ) {
            elog( "database metadata dirty flag set (likely due to unclean
                shutdown): replay required" );
            return DATABASE_DIRTY;
        } else {
            elog( "${e}", ("e",e.what()));
        }
        return OTHER_FAIL;
    } catch( const std::exception& e ) {
        elog("${e}", ("e",e.what()));
        return OTHER_FAIL;
    } catch( ... ) {
        elog("unknown exception");
        return OTHER_FAIL;
    }

    return SUCCESS;
}
```

其中核心代码在 initialize() 函数由引入了四个插件，这里初始化使用了 chain_plugin（区块链插件）、http_plugin（HTTP 插件，和 cleos 交互）、net_plugin（P2P 网络插件）、producer_plugin（生产者插件）这四个插件。initialize() 函数通过调用 find_plugin() 实现根据插件名找到插件；然后把找到的插件（数组）传递给 application::initialize_impl() 函数进行初始化；startup() 函数作为 plugin 超类的抽象函数，通过 initialize() 函数初始化后遍历插件数组（initialized_plugins），然后调用每个 plugin 的 starup() 函数；通过 exec() 函数调用 boost 库的 io_context::run() 函数，从 io_context 的名字中就可以看出，其是“输入 / 输出上下文”的意思。最终会调用 boost 库中的 scheduler::run() 函数，它会开启线程，运行一个异步的 IO 服务。这样服务端就可以一直运行了。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

3.6.2 cleos 代码实现

cleos 就是 client eos，它是一个标准的客户端程序，具备如下特点：它是一个标准的命令行解析器，类似于 Linux 操作系统的终端；它同样具有强大的扩展功能；它不存在数据库，它需要的所有数据都是通过 HTTP 协议去连接 nodeos 和 keosd（这两个进程都有 HTTP 的监听端口）的，通过反射机制拿到自己想要的数据再进行逻辑运算并把执行结果显示出来。cleos 中 main 配置了一些文件路径并初始化了数据的读取。cleos 中所包含的函数如图 3-26 所示。

下面我们来一起查看 eos/programs/cleos/main.cpp 文件，并对上述 cleos 涉及的命令进行简单说明。在介绍子命令的之前，我们先看看分析三个，即 add_subcommand()、add_option() 和 set_callback() 函数。在 eos/programs/cleos/CLI11.hpp 定义了这些函数。需要注意的是，add_subcommand() 函数与 add_option() 函数返回的变量都是 vector 类型。

首先来看 version 命令，在这里我们不难看其是通过 App 的成员函数 add_sub command() 成功实现了链表，并引入了 version 命令。version 对象再调用 add_subcommand() 函数从而引入了子命令 client，这样就获取到了本地 EOSIO 客户端的版本信息。相应代码如下：

```
//请求EOSIO客户端加载内置类APP，并对其初始化
CLI::App app{"Command Line Interface to EOSIO Client"};
    app.require_subcommand();
//添加version子命令
auto version = app.add_subcommand("version", localized("Retrieve version
information"), false);
    version->require_subcommand();
//添加version的子命令client
    version->add_subcommand("client", localized("Retrieve version information
        of the client"))->set_callback([] {
        std::cout << localized("Build version: ${ver}", ("ver", eosio::client::
            config::version_str)) << std::endl;
    });
```

接着来看 create 命令，其作用是在 EOS 区块链上创建账户密钥。其子命令 key 用于创建一对公私钥对，当添加“--rl”非必要参数时表示创建基于 rl 曲线生成的公私钥对。子命令 account 用于创建一个新账户。相应代码如下：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

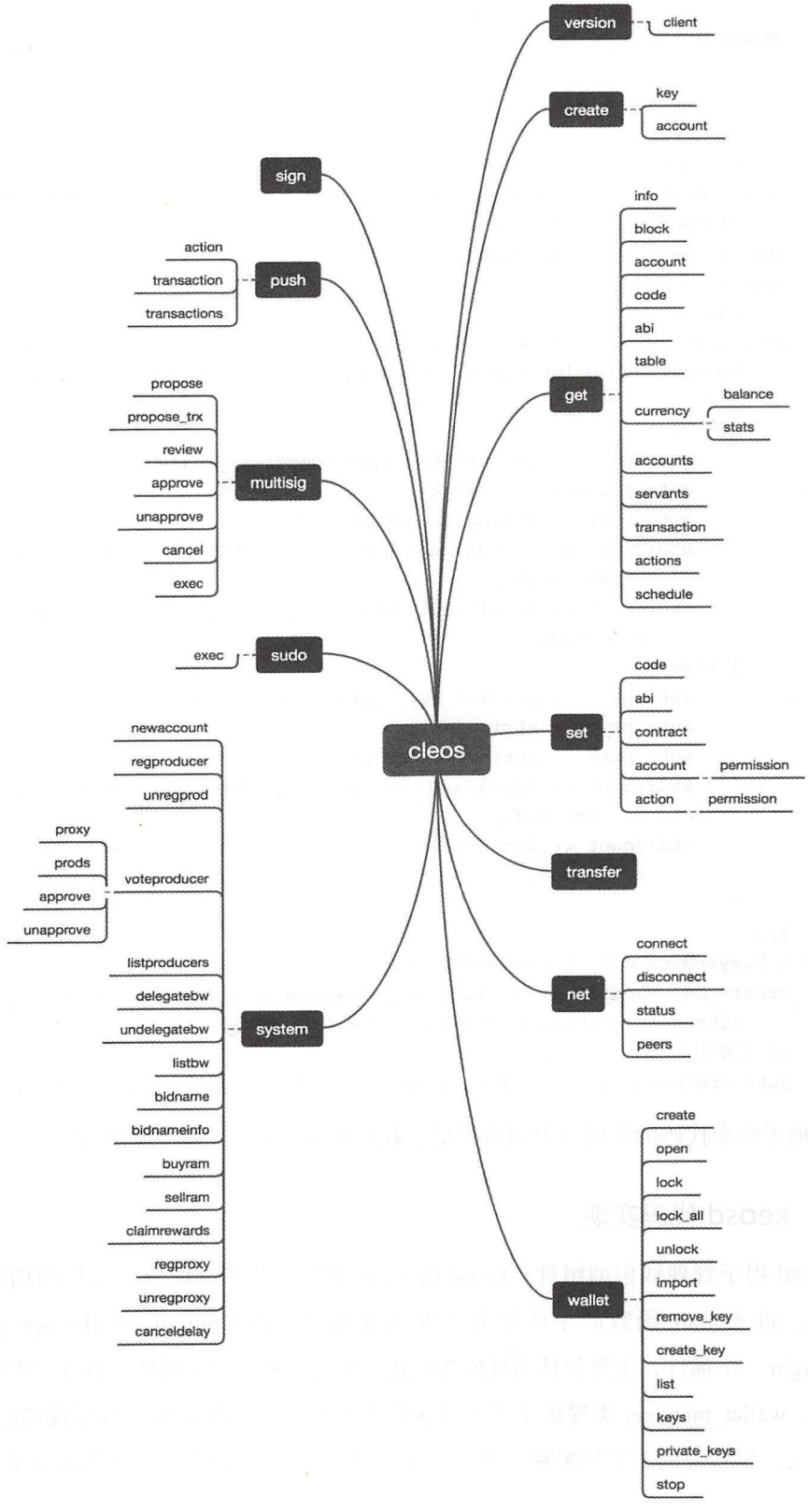


图 3-26　cleos 中所包含的函数

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    // 添加 create 子命令
    auto create = app.add_subcommand("create", localized("Create various
        items, on and off the blockchain"), false);
    create->require_subcommand();
    bool r1 = false;
    // 添加create的子命令 key
    auto create_key = create->add_subcommand("key", localized("Create a new
        keypair and print the public and private keys"))->set_callback( [&r1]
        (){
        if( r1 ) {
            auto pk    = private_key_type::generate_r1();
            auto privs = string(pk);
            auto pubs  = string(pk.get_public_key());
            std::cout << localized("Private key: ${key}", ("key",  privs) )
                << std::endl;
            std::cout << localized("Public key: ${key}", ("key", pubs ) ) <<
                std::endl;
        } else {
            auto pk    = private_key_type::generate();
            auto privs = string(pk);
            auto pubs  = string(pk.get_public_key());
            std::cout << localized("Private key: ${key}", ("key",  privs) )
                << std::endl;
            std::cout << localized("Public key: ${key}", ("key", pubs ) ) <<
                std::endl;
        }
    });
//在命令key的情况下添加一个非必需的参数--r1
    create_key->add_flag( "--r1", r1, "Generate a key using the R1 curve
        (iPhone), instead of the K1 curve (Bitcoin)" );
    // 创建账户
    auto createAccount = create_account_subcommand( create, true /*simple*/ );
```

其他子命令代码的作用与上面的类似，我们就不对其一一详细说明了。

3.6.3 keosd 代码实现

keosd 用于管理钱包的组件。keosd 的 main 函数开头部分配置了文件的创建与加载命令，进入 main 函数的主体部分会发现加载了 wallet_plugin、wallet_api_plugin、http_plugin 三个插件，主要包括插件的初始化、运行、停止三个操作。打开三个插件就会发现，wallet_manager 类提供了关于钱包的所有操作的 API 函数，包括钱包的创建、密匙导入、公匙查询、加锁解锁、私匙签名等 API。main 函数的实现代码如下所示：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
int main(int argc, char** argv)
{
    try {
        bfs::path home = determine_home_directory();
        app().set_default_data_dir(home / "eosio-wallet");
        app().set_default_config_dir(home / "eosio-wallet");
        app().register_plugin<wallet_api_plugin>();
        if(!app().initialize<wallet_plugin, wallet_api_plugin, http_plugin>
            (argc, argv))
            return -1;
        auto& http = app().get_plugin<http_plugin>();
        http.add_handler("/v1/keosd/stop", [](string, string, url_response_
            callback cb) { cb(200, "{}"); std::raise(SIGTERM); } );
        app().startup();
        app().exec();
    } catch (const fc::exception& e) {
        elog("${e}", ("e",e.to_detail_string()));
    } catch (const boost::exception& e) {
        elog("${e}", ("e",boost::diagnostic_information(e)));
    } catch (const std::exception& e) {
        elog("${e}", ("e",e.what()));
    } catch (...) {
        elog("unknown exception");
    }
    return 0;
}
```

wallet_manager 类的实现代码如下所示：

```
//引入文件
#include <eosio/wallet_plugin/wallet_manager.hpp>
#include <eosio/wallet_plugin/wallet.hpp>
#include <eosio/wallet_plugin/se_wallet.hpp>
#include <eosio/chain/exceptions.hpp>
#include <boost/algorithm/string.hpp>
namespace eosio {
namespace wallet {
//定义两个变量
constexpr auto file_ext = ".wallet";
constexpr auto password_prefix = "PW";
//定义key的生成格式
std::string gen_password() {
    auto key = private_key_type::generate();
    return password_prefix + string(key);
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
}
//验证是否为有效的文件名
bool valid_filename(const string& name) {
    if (name.empty()) return false;
    if (std::find_if(name.begin(), name.end(), !boost::algorithm::is_alnum()
        && !boost::algorithm::is_any_of("._-")) != name.end()) return false;
    return boost::filesystem::path(name).filename().string() == name;
}
//查看定义的钱包名是否是唯一的
wallet_manager::wallet_manager() {
#ifdef __APPLE__
    try {
        wallets.emplace("SecureEnclave", std::make_unique<se_wallet>());
    } catch(fc::exception& ) {}
#endif
}

//设置钱包被锁定的时间
void wallet_manager::set_timeout(const std::chrono::seconds& t) {
    timeout = t;
    auto now = std::chrono::system_clock::now();
    timeout_time = now + timeout;
    EOS_ASSERT(timeout_time >= now, invalid_lock_timeout_exception, "Overflow
        on timeout_time, specified ${t}, now ${now}, timeout_time ${timeout_
        time}",
            ("t", t.count())("now", now.time_since_ePoCh().count())("timeout_
                time", timeout_time.time_since_ePoCh().count()));
}
//检测钱包的锁定时间
void wallet_manager::check_timeout() {
    if (timeout_time != timepoint_t::max()) {
        const auto& now = std::chrono::system_clock::now();
        if (now >= timeout_time) {
            lock_all();
        }
        timeout_time = now + timeout;
    }
}
//创建新的钱包
std::string wallet_manager::create(const std::string& name) {
    check_timeout();
//调用函数valid_filename()
    EOS_ASSERT(valid_filename(name), wallet_exception, "Invalid filename,
        path not allowed in wallet name ${n}", ("n", name));
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    auto wallet_filename = dir / (name + file_ext);

    if (fc::exists(wallet_filename)) {
        EOS_THROW(chain::wallet_exist_exception, "Wallet with name: '${n}'
            already exists at ${path}", ("n", name)("path",fc::path(wallet_
            filename)));
    }

    std::string password = gen_password();
    wallet_data d;
    auto wallet = make_unique<soft_wallet>(d);
    wallet->set_password(password);
    wallet->set_wallet_filename(wallet_filename.string());
    wallet->unlock(password);
    wallet->lock();
    wallet->unlock(password);

    // 在此处明确保存钱包文件，以确保它存在
    wallet->save_wallet_file();

    //如果我们在地图中有名字，那么删除它，因为我们希望进行下面的安置替换
    //如果在eos-walletd运行时删除了钱包文件，则会发生这种情况
    auto it = wallets.find(name);
    if (it != wallets.end()) {
        wallets.erase(it);
    }
    wallets.emplace(name, std::move(wallet));

    return password;
}
//打开未上锁的钱包
void wallet_manager::open(const std::string& name) {
    check_timeout();

    EOS_ASSERT(valid_filename(name), wallet_exception, "Invalid filename,
        path not allowed in wallet name ${n}", ("n", name));

    wallet_data d;
    auto wallet = std::make_unique<soft_wallet>(d);
    auto wallet_filename = dir / (name + file_ext);
    wallet->set_wallet_filename(wallet_filename.string());
    if (!wallet->load_wallet_file()) {
        EOS_THROW(chain::wallet_nonexistent_exception, "Unable to open file:
            ${f}", ("f", wallet_filename.string()));
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    }

    //如果我们在地图中有名字，那么删除它，因为我们希望进行下面的安置替换
    //如果在eos-walletd运行时删除了钱包文件，则会发生这种情况
    auto it = wallets.find(name);
    if (it != wallets.end()) {
        wallets.erase(it);
    }
    wallets.emplace(name, std::move(wallet));
}
//获取所有的钱包列表，未上锁的加上‘*’进行标注
std::vector<std::string> wallet_manager::list_wallets() {
    check_timeout();
    std::vector<std::string> result;
    for (const auto& i : wallets) {
        if (i.second->is_locked()) {
            result.emplace_back(i.first);
        } else {
            result.emplace_back(i.first + " *");
        }
    }
    return result;
}
//查看钱包中的keys
map<public_key_type,private_key_type> wallet_manager::list_keys(const
    string& name, const string& pw) {
    check_timeout();

    if (wallets.count(name) == 0)
        EOS_THROW(chain::wallet_nonexistent_exception, "Wallet not found:
            ${w}", ("w", name));
    auto& w = wallets.at(name);
    if (w->is_locked())
        EOS_THROW(chain::wallet_locked_exception, "Wallet is locked: ${w}",
            ("w", name));
    w->check_password(pw); //throws if bad password
    return w->list_keys();
}
//获取未上锁钱包中导入的公钥
flat_set<public_key_type> wallet_manager::get_public_keys() {
    check_timeout();
    EOS_ASSERT(!wallets.empty(), wallet_not_available_exception, "You don't
        have any wallet!");
    flat_set<public_key_type> result;
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

```
    bool is_all_wallet_locked = true;
    for (const auto& i : wallets) {
        if (!i.second->is_locked()) {
            result.merge(i.second->list_public_keys());
        }
        is_all_wallet_locked &= i.second->is_locked();
    }
    EOS_ASSERT(!is_all_wallet_locked, wallet_locked_exception, "You don't
        have any unlocked wallet!");
    return result;
}

//为全部钱包上锁
void wallet_manager::lock_all() {
    // 因为要锁定所有钱包，所以没有调用check_timeout
    for (auto& i : wallets) {
        if (!i.second->is_locked()) {
            i.second->lock();
        }
    }
}
//给指定的钱包上锁
void wallet_manager::lock(const std::string& name) {
    check_timeout();
    if (wallets.count(name) == 0) {
        EOS_THROW(chain::wallet_nonexistent_exception, "Wallet not found:
            ${w}", ("w", name));
    }
    auto& w = wallets.at(name);
    if (w->is_locked()) {
        return;
    }
    w->lock();
}
//为上锁的钱包开锁
void wallet_manager::unlock(const std::string& name, const std::string& password) {
    check_timeout();
    if (wallets.count(name) == 0) {
        open( name );
    }
    auto& w = wallets.at(name);
if (!w->is_locked()) {
        EOS_THROW(chain::wallet_unlocked_exception, "Wallet is already
            unlocked: ${w}", ("w", name));
```

非卖品，仅供非商业用途或交流学习使用


```
        return;
    }
    w->unlock(password);
}
//为钱包导入存在的key
void wallet_manager::import_key(const std::string& name, const std::string& wif_
    key) {
    check_timeout();
    if (wallets.count(name) == 0) {
        EOS_THROW(chain::wallet_nonexistent_exception, "Wallet not found:
            ${w}", ("w", name));
    }
    auto& w = wallets.at(name);
    if (w->is_locked()) {
        EOS_THROW(chain::wallet_locked_exception, "Wallet is locked: ${w}",
            ("w", name));
    }
    w->import_key(wif_key);
}
//删除钱包中的密钥
void wallet_manager::remove_key(const std::string& name, const std::string&
    password, const std::string& key) {
    check_timeout();
    if (wallets.count(name) == 0) {
        EOS_THROW(chain::wallet_nonexistent_exception, "Wallet not found:
            ${w}", ("w", name));
    }
    auto& w = wallets.at(name);
    if (w->is_locked()) {
        EOS_THROW(chain::wallet_locked_exception, "Wallet is locked: ${w}",
            ("w", name));
    }
    w->check_password(password); //throws if bad password
    w->remove_key(key);
}
//为钱包创建key
string wallet_manager::create_key(const std::string& name, const std::
    string& key_type) {
    check_timeout();
    if (wallets.count(name) == 0) {
        EOS_THROW(chain::wallet_nonexistent_exception, "Wallet not found:
            ${w}", ("w", name));
    }
    auto& w = wallets.at(name);
```

```
    if (w->is_locked()) {
        EOS_THROW(chain::wallet_locked_exception, "Wallet is locked: ${w}",
            ("w", name));
    }

    string upper_key_type = boost::to_upper_copy<std::string>(key_type);
    return w->create_key(upper_key_type);
}

chain::signed_transaction
wallet_manager::sign_transaction(const chain::signed_transaction& txn, const
    flat_set<public_key_type>& keys, const chain::chain_id_type& id) {
    check_timeout();
    chain::signed_transaction stxn(txn);

    for (const auto& pk : keys) {
        bool found = false;
        for (const auto& i : wallets) {
            if (!i.second->is_locked()) {
                optional<signature_type> sig = i.second->try_sign_
                    digest(stxn.sig_digest(id, stxn.context_free_data), pk);
                if (sig) {
                    stxn.signatures.push_back(*sig);
                    found = true;
                    break; // inner for
                }
            }
        }
        if (!found) {
            EOS_THROW(chain::wallet_missing_pub_key_exception, "Public key
                not found in unlocked wallets ${k}", ("k", pk));
        }
    }

    return stxn;
}

chain::signature_type
wallet_manager::sign_digest(const chain::digest_type& digest, const public_
    key_type& key) {
    check_timeout();
    try {
        for (const auto& i : wallets) {
            if (!i.second->is_locked()) {
```

```
                optional<signature_type> sig = i.second->try_sign_digest(digest, key);
                if (sig)
                    return *sig;
            }
        }
    } FC_LOG_AND_RETHROW();

    EOS_THROW(chain::wallet_missing_pub_key_exception, "Public key not found
        in unlocked wallets ${k}", ("k", key));
}

void wallet_manager::own_and_use_wallet(const string& name, std::unique_
    ptr<wallet_api>&& wallet) {
    if(wallets.find(name) != wallets.end())
        FC_THROW("tried to use wallet name the already existed");
    wallets.emplace(name, std::move(wallet));
}

} // namespace wallet
} // namespace eosio
```

第二部分 Part 2

实 战 篇

- 第 4 章　EOS 网络启动与原生功能介绍
- 第 5 章　智能合约开发
- 第 6 章　基于 EOS 的 DApp 开发
- 第 7 章　基于 EOS 的侧链 / 竞争链开发

Chapter 4 第 4 章

EOS 网络启动与原生功能介绍

本章主要介绍如何在个人电脑上搭建 EOS 本地环境，并在此环境下如何运行测试节点、项目原生功能，以及智能合约的部署和调用。

4.1 EOS 项目安装部署与编译

目前网络上很多关于 EOS 编译安装的文章都是基于老版本的源码，根据这些文章需要手动安装 EOS 需要的库，而本节是基于 EOS 项目的最新源码展开的，并不需要自己手动去安装每一个库。

运行 EOS 源码对系统的要求是内存不小于 7GB，系统存储空间不小于 20GB，因此在搭建系统时，要尽量多分配一些内存与存储空间。当然如果已安装的系统无法满足此要求，后面编译的时候就可以通过注释代码的形式绕过去，这不会影响编译效果。

1. 构建 EOS

打开 Ubuntu 命令窗口，输入以下命令，按下 Enter 键，系统会在 home 目录下自动生成 EOS 文件夹并且下载 EOS 源码，-recursive 可以确保你下载了属于这个项目的所有“东西”，它会层级式地复制这个目录，一直复制到目录树的最底层。

```
git clone https://github.com/EOSIO/eos -recursive
```

这里下载的 EOS 版本为 v1.1.0，在下载过程中，如果网络偏慢，可在终端输入 sudo vim /etc/hosts 命令，这样可以用 vim 打开 host 文件，通过输入 i 字符打开插入模式，在 IP 地址中添加 192.30.253.113 github.com，而后保存该文件。该操作可以提升 github 网站下载速度。

2. 编译

为了搭建 EOS，我们需要使用刚刚下载的文件夹里的一个文件——eosio_build.sh。以“.sh”结尾说明这是一个 shell 脚本。一句话概括就是：这是一个可以直接通过命令行执行的文件。在终端进入 EOS 文件夹后，输入如下命令：

```
./eosio_build.sh
```

执行这个脚本需要花一段时间，并且它很有可能会失败一两次。下面列出了几个常见的报错问题及其解决办法。

（1）电脑内存小于 7GB 或者存储空间小于 20GB 报错。

在 Ubuntu 平台下，找到 ./scripts/eosio_build_ubuntu.sh 文件，将图 4-1 和图 4-2 中所示的“exit 1”注释掉即可。

```
27      if [ "${MEM_MEG}" -lt 7000 ]; then
28          printf "\\tYour system must have 7 or more Gigabytes of physical memory installed.\\n"
29          printf "\\tExiting now.\\n"
30          exit 1
```

图 4-1　电脑内存检测位置

```
57      if [ "${DISK_AVAIL%.*}" -lt "${DISK_MIN}" ]; then
58          printf "\\tYou must have at least %sGB of available storage to install EOSIO.\\n" "${DISK_MIN}"
59          printf "\\tExiting now.\\n"
60          //exit 1
```

图 4-2　电脑存储检测位置

（2）在安装 secp256k1-zkp 和 LLVM 时提示安装失败，make file 参数错误。

找到 ./scripts/eosio_build_ubuntu.sh 文件，根据自己电脑核心数将 make -j “${JOBS}” install 改为 make -j install 即可（其中 j 表示电脑核心数）。

最后编译成功会显示图 4-3 所示的信息。

EOS 编译完之后，切换到 eos/build 文件夹，安装 EOS，执行以下命令：

```
cd build
sudo make install
```

图 4-3 EOS 编译成功界面

以上命令会自动安装 EOS，安装成功后，会在 build 目录下生成安装目录。在 usr/local/bin 目录下有所有编译好的命令，包括 cleos、nodeos、eosiocpp 等。

注意，新版本编译命令在 usr/local/eosio/bin 文件中，需要执行以下命令：

```
vi ~/.bashrc
```

进入上述文件插入以下内容：

```
export PATH=/usr/local/eosio/bin: $PATH
export LD_LIBRARY_PATH=/usr/local/eosio/lib: $LD_LIBRARY_PATH
```

保存退出之后，我们就可以在全局命令中使用 cleos、nodeos、eoscpp 等命令了。

4.2 配置文件修改与测试

EOS 主要有三个程序——cloes、nodeos、keosd，在 eos/build/programs 文件夹下，如图 4-4 所示，三个程序的主要功能如下。

- cloes：客户端命令行交互模块，用于解析用户命令，根据具体命令请求调用相应的接口，例如查看区块信息、操作钱包等。

非卖品，仅供非商业用途或交流学习使用

- nodeos：服务器端，也就是区块生产节点，用于接收客户端的远端请求，并打包区块，主要包含 chain_plugin、http_plugin、net_plugin、producer_plugin 四个插件。
- keosd：钱包管理模块，主要包括 wallet_plugin、wallet_api_plugin、http_plugin 三个插件。

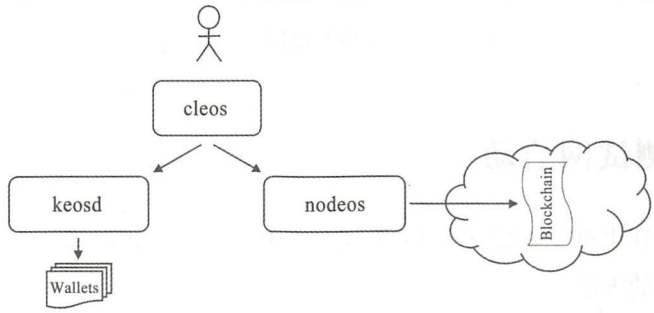


图 4-4　cloes、nodeos、keosd 三程序的关系

进入 build/programs/nodeos 目录，启动 nodeos 的命令如下：

```
./nodeos -e -p eosio --plugin eosio::chain_api_plugin --plugin eosio::history_
    api_plugin
```

如果不带参数执行 nodeos 程序，会报错，这是因为没有为 nodeos 程序配置各种插件，需要修改 config 文件给 nodeos 配置插件。输入以下命令：

```
vi ~/.local/share/eosio/nodeos/config/config.ini
```

修改以下内容：

```
enable-stale-production = true
producer-name = eosio
```

并在文档末尾添加内容：

```
plugin = eosio::producer_plugin
plugin = eosio::wallet_api_plugin
plugin = eosio::chain_api_plugin
plugin = eosio::account_history_api_plugin
```

重新启动 nodeos 程序，会显示图 4-5 所示的信息，表明本地测试网上已经成功挖出区块了。

非卖品，仅供非商业用途或交流学习使用


```
2018-08-08T07:33:17.001 thread-0   producer_plugin.cpp:1235      produce_block        ] Produced
 block 000099271f0d6b68... #39207 @ 2018-08-08T07:33:17.000 signed by eosio [trxs: 0, lib: 39206
, confirmed: 0]
2018-08-08T07:33:17.501 thread-0   producer_plugin.cpp:1235      produce_block        ] Produced
 block 0000992823557a12... #39208 @ 2018-08-08T07:33:17.500 signed by eosio [trxs: 0, lib: 39207
, confirmed: 0]
2018-08-08T07:33:18.001 thread-0   producer_plugin.cpp:1235      produce_block        ] Produced
 block 00009929d16b8c65... #39209 @ 2018-08-08T07:33:18.000 signed by eosio [trxs: 0, lib: 39208
, confirmed: 0]
2018-08-08T07:33:18.501 thread-0   producer_plugin.cpp:1235      produce_block        ] Produced
 block 0000992a34db9f75... #39210 @ 2018-08-08T07:33:18.500 signed by eosio [trxs: 0, lib: 39209
, confirmed: 0]
```

图 4-5 在本地测试网挖出区块

4.3 EOS 测试网启动

本节将介绍在电脑上创建 3 个 EOS 节点（Bios 节点、预出块节点、普通节点）并组成一个私有链的方法。

首先我们新建三个文件夹，分别命名为 bios、producer、nod。每个文件夹下都新建配置文件 config.ini 和创世文件 genesis.json。其中三个文件中的 genesis.json 都相同，内容为：

```
{
    "initial_timestamp": "2018-03-01T12:00:00.000",
    "initial_key": "EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV",
    "initial_configuration": {
        "base_per_transaction_net_usage": 100,
        "base_per_transaction_cpu_usage": 500,
        "base_per_action_cpu_usage": 1000,
        "base_setcode_cpu_usage": 2097152,
        "per_signature_cpu_usage": 100000,
        "per_lock_net_usage": 32,
        "context_free_discount_cpu_usage_num": 20,
        "context_free_discount_cpu_usage_den": 100,
        "max_transaction_cpu_usage": 10485760,
        "max_transaction_net_usage": 104857,
        "max_block_cpu_usage": 104857600,
        "target_block_cpu_usage_pct": 1000,
        "max_block_net_usage": 1048576,
        "target_block_net_usage_pct": 1000,
        "max_transaction_lifetime": 3600,
        "max_transaction_exec_time": 0,
        "max_authority_depth": 6,
        "max_inline_depth": 4,
        "max_inline_action_size": 4096,
        "max_generated_transaction_count": 16
```

非卖品，仅供非商业用途或交流学习使用


```
    },
    "initial_chain_id": "0000000000000000000000000000000000000000000000000000000000000000"
}
```

4.3.1　Bios 节点

对 bios 文件夹下的 config.ini 文件写入以下内容：

```
genesis-json = genesis.json
block-log-dir = blocks
readonly = 0
send-whole-blocks = true
shared-file-dir = blockchain
shared-file-size = 8192
unlock-timeout = 90000
bnet-endpoint = 0.0.0.0:4321
http-server-address = 0.0.0.0:8888
p2p-listen-endpoint = 0.0.0.0:9876
p2p-server-address = localhost:9876
allowed-connection = any
#p2p-peer-address = localhost:9877
required-participation = true
signature-provider = EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5G
DW5CV=KEY:5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3
producer-name = eosio
plugin = eosio::producer_plugin
plugin = eosio::chain_api_plugin
plugin = eosio::history_api_plugin
plugin = eosio::wallet_api_plugin
plugin = eosio::http_plugin
enable-stale-production = true
```

在终端进入 bios 目录后，使用以下命令启动 bios 节点并让它出块：

```
$ sudo nodeos --data-dir ./ --config-dir ./
```

bios 节点启动之后，还需要用以下命令发布 eosio.bios 合约：

```
cleos set contract eosio eos/build/contracts/eosio.bios/
```

4.3.2　预出块节点

在启动预出块节点之前，我们需要创建该节点的账户，用以下命令创建 produce 账户，如图 4-6 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
cleos create account eosio peoduce EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHu
    GYqET5GDW5CV
```

```
guo@guo-XPS-13-9360:~/eos$ cleos create account eosio produce EOS6MRyAjQq8ud7hVNYcfnVPJqc
VpscN5So8BhtHuGYqET5GDW5CV
executed transaction: f26b5f4924057590e7829935c0dcef4b1774be737066b0ea3635e2d9c8894114  2
00 bytes  25967 us
#         eosio <= eosio::newaccount           {"creator":"eosio","name":"produce","owne
r":{"threshold":1,"keys":[{"key":"EOS6MRyAjQq8ud7hVNYcfnVPJ...
2018-08-24T05:19:05.895 thread-0   main.cpp:432                 print_result   warning:
transaction executed locally, but may not be confirmed by the network yet
```

图 4-6 produce 账号创建

对 produce 文件夹下的 config.ini 文件写入以下内容：

```
genesis-json = genesis.json
block-log-dir = blocks
readonly = 0
send-whole-blocks = true
shared-file-dir = blockchain
shared-file-size = 8192
unlock-timeout = 90000
bnet-endpoint = 0.0.0.0:4321
http-server-address = 0.0.0.0:8889
p2p-listen-endpoint = 0.0.0.0:9877
p2p-server-address = localhost:9877
allowed-connection = any
p2p-peer-address = localhost:9876
required-participation = true
signature-provider = EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GD W5CV=
    KEY:5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3
producer-name = produce
plugin = eosio::producer_plugin
plugin = eosio::chain_api_plugin
plugin = eosio::history_api_plugin
plugin = eosio::wallet_api_plugin
plugin = eosio::http_plugin
#enable-stale-production = true
```

其中 p2p-peer-address = localhost:9876，这正是 bios 节点上的 P2P 服务地址，这将使得出块节点可以连接上 bios 节点并进行同步。通过 producer-name = produce 创建预出块节点账号名。

输入以下命令启动预出块节点：

```
sudo nodeos --data-dir ./ --config-dir ./
```

此时预出块节点只是连接上 bios 节点，还不能出块。使用以下命令配置 produce

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

出块，如图 4-7 所示。

```
cleos push action eosio setprods '{"schedule":[{"producer_name":"produce",
"block_signing_key":"EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5
    CV"}]}' -p eosio
```

```
guo@guo-XPS-13-9360:~/eos$ cleos push action eosio setprods '{"schedule":[{"producer_name
":"produce","block_signing_key":"EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW5CV"}]
}' -p eosio
executed transaction: 1a43c9e73707cee67223e800a0ad761adf171748367f92c2b1e878db7e37b0de  1
36 bytes  642 us
#         eosio <= eosio::setprods              {"schedule":[{"producer_name":"produce","
block_signing_key":"EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8Bht...
2018-08-24T05:19:28.934 thread-0   main.cpp:432                 print_result   warning:
transaction executed locally, but may not be confirmed by the network yet
```

图 4-7　produce 出块节点命令配置

此时 produce 变成出块节点，而 bios 节点成了同步节点，如图 4-8 和图 4-9 所示。

```
guo@guo-XPS-13-9360: ~/eos/testnet/produce
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
2018-08-24T05:24:58.000 thread-0   producer_plugin.cpp:1235      produce_block        ] P
roduced block 00000bce55fda1e6... #3022 @ 2018-08-24T05:24:58.000 signed by produce [trxs
: 0, lib: 3021, confirmed: 0]
2018-08-24T05:24:58.500 thread-0   producer_plugin.cpp:1235      produce_block        ] P
roduced block 00000bcf66a41b43... #3023 @ 2018-08-24T05:24:58.500 signed by produce [trxs
: 0, lib: 3022, confirmed: 0]
2018-08-24T05:24:59.001 thread-0   producer_plugin.cpp:1235      produce_block        ] P
roduced block 00000bd0e5f09843... #3024 @ 2018-08-24T05:24:59.000 signed by produce [trxs
: 0, lib: 3023, confirmed: 0]
```

图 4-8　produce 节点出块

```
guo@guo-XPS-13-9360: ~/eos/testnet/bios
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
ceived block 4e759835e6c1abf4... #3015 @ 2018-08-24T05:24:54.500 signed by produce [trxs:
0, lib: 3014, conf: 0, latency: 27 ms]
2018-08-24T05:24:55.002 thread-0   producer_plugin.cpp:331       on_incoming_block    ] Re
ceived block 382a80ac05e7511b... #3016 @ 2018-08-24T05:24:55.000 signed by produce [trxs:
0, lib: 3015, conf: 0, latency: 2 ms]
2018-08-24T05:24:55.502 thread-0   producer_plugin.cpp:331       on_incoming_block    ] Re
ceived block be61957515036152... #3017 @ 2018-08-24T05:24:55.500 signed by produce [trxs:
0, lib: 3016, conf: 0, latency: 2 ms]
```

图 4-9　bios 节点收块

4.3.3　普通节点

在普通节点启动前同样需要创建一个账号，使用上节命令创建账户 nod，并在 nod

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

文件夹下的 config.ini 文件中写入以下内容：

```
genesis-json = genesis.json
block-log-dir = blocks
readonly = 0
send-whole-blocks = true
shared-file-dir = blockchain
shared-file-size = 8192
unlock-timeout = 90000
bnet-endpoint = 0.0.0.0:4321
http-server-address = 0.0.0.0:8890
p2p-listen-endpoint = 0.0.0.0:9878
p2p-server-address = localhost:9878
allowed-connection = any
p2p-peer-address = localhost:9877
required-participation = true
signature-provider = EOS6MRyAjQq8ud7hVNYcfnVPJqcVpscN5So8BhtHuGYqET5GDW 5CV=
    KEY:5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3
producer-name = nod
plugin = eosio::producer_plugin
plugin = eosio::chain_api_plugin
plugin = eosio::history_api_plugin
plugin = eosio::wallet_api_plugin
plugin = eosio::http_plugin
#enable-stale-production = true
```

输入以下命令启动该节点：

```
sudo nodeos --data-dir ./ --config-dir ./
```

该节点会同步区块信息。

4.4 EOS 项目常用功能

4.4.1 链上信息查询

本小节主要讲解连接上主网后，如何通过 cleos get 命令查询主网上的信息。

1. 确认程序是否已连接 EOS 主网

首先我们得确认我们的程序是否已连接 EOS 主网。运行 nodeos 后，我们在命令行中输入如下命令：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
cleos get info
```

此时出现图 4-10 所示的信息。

```
➜ eos git:(master) cleos get info
{
  "server_version": "5875549c",
  "chain_id": "cf057bbfb72640471fd910bcb67639c22df9f92470936cddc1ade0e2f2e7dc4f",
  "head_block_num": 6878,
  "last_irreversible_block_num": 6877,
  "last_irreversible_block_id": "00001adde53f8c31c9dd3808a8441980915165384914ee31871713076e60ba31",
  "head_block_id": "00001ade1be8798b199bf51ba541641068ba88dc067d2223ec8784ceda51820e",
  "head_block_time": "2018-07-07T11:38:17.000",
  "head_block_producer": "eosio",
  "virtual_block_cpu_limit": 194122639,
  "virtual_block_net_limit": 1019840961,
  "block_cpu_limit": 199900,
  "block_net_limit": 1048576
}
```

chain_id

图 4-10　查询所连接区块链的信息

注意，上图所示界面里面有一个 chain_id 字段，它的值是 cf057...c4f，这是它所在网络的标识符，而主网的 chain_id 的值是 aca376f206b8fc25a6ed44dbdc66547c36c6c33e3a119ffbeaef943642f0e9 06。很明显，我们并没有连接到主网中，因此我们还需要通过其他方法连接到主网。

在运行 nodeos 时需要指定的 genesis.json 文件，在主目录下新创建一个 genesis.json 文件，并把下面的内容复制并粘贴进去。

```
{
"initial_timestamp": "2018-06-08T08:08:08.888",
"initial_key": "EOS7EarnUhcyYqmdnPon8rm7mBCTnBoot6o7fE2WzjvEX2TdggbL3",
"initial_configuration": {
    "max_block_net_usage": 1048576,
    "target_block_net_usage_pct": 1000,
    "max_transaction_net_usage": 524288,
    "base_per_transaction_net_usage": 12,
    "net_usage_leeway": 500,
    "context_free_discount_net_usage_num": 20,
    "context_free_discount_net_usage_den": 100,
    "max_block_cpu_usage": 200000,
    "target_block_cpu_usage_pct": 1000,
    "max_transaction_cpu_usage": 150000,
    "min_transaction_cpu_usage": 100,
    "max_transaction_lifetime": 3600,
    "deferred_trx_expiration_window": 600,
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    "max_transaction_delay": 3888000,
    "max_inline_action_size": 4096,
    "max_inline_action_depth": 4,
    "max_authority_depth": 6
    }
}
```

指定运行时的插件：

```
Macos: "~ /Library/Application/Support/eosio/nodeos/config/config.ini"
Ubuntu: "~/.local/share/eosio/nodeos/config"
```

在上述文件中添加插件列表：

```
plugin = eosio::chain_plugin
plugin = eosio::chain_api_plugin
plugin = eosio::net_plugin
plugin = eosio::net_api_plugin
plugin = eosio::history_plugin
plugin = eosio::history_api_plugin
plugin = eosio::http_plugin
plugin = eosio::http_client_plugin
plugin = eosio::wallet_api_plugin
```

之后我们再运行 nodeos --genesis-json ~/genesis.json，这时又遇到了一个问题，如图 4-11 所示。

```
2964841ms thread-0   main.cpp:113                    main                ] 10 assert_exception: Assert Exception
!fc::exists( my->blocks_dir / "blocks.log" ): Genesis state can only be set on a fresh blockchain.
    {}
    thread-0  chain_plugin.cpp:366 plugin_initialize
```

图 4-11 连接主网数据不匹配错误

这是由于之前 nodeos 运行测试网时留下的区块数据不匹配，我们只需要把之前的数据删除即可：

```
rm -rf  ~/Library/Application\ Support/eosio/nodeos/data
```

或者在启动 nodeos 时追加 --delete-all-blocks 参数。

```
nodeos --genesis-json ~/genesis.json --delete-all-blocks
```

经过上述操作，运行 cleos get info 之后出来的也是主网 ID，一切看上去都是正常的。但是为什么我电脑只有一个块？且执行 cleos get block 2 就会报错。节点要同步数据，就必须知道从哪个节点同步数据。这需要显式声明种子节点来解决，这些种子节

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

点通常是 EOS 的创建者和运营者。同样，我们在之前的 config.ini 文件中继续追加种子节点信息：

```
p2p-peer-address = p2p.prod.eosgravity.com:80
p2p-peer-address = eu-west-nl.eosamsterdam.net:9876
p2p-peer-address = p2p.mainnet.eosgermany.online:9876
p2p-peer-address = 54.38.250.15:19878
p2p-peer-address = p2p.genereos.io:9876
p2p-peer-address = mainnet.eospay.host:19876
p2p-peer-address = 130.211.59.178:9876
p2p-peer-address = 54.153.59.31:9999
p2p-peer-address = 94.130.250.22:9806
p2p-peer-address = peer.main.alohaeos.com:9876
p2p-peer-address = peer.eosn.io:9876
p2p-peer-address = prod.mainnet.eos.cybex.io:9888
p2p-peer-address = p2p-1.eosnetwork.io:9876
p2p-peer-address = p.jeda.one:3322
p2p-peer-address = eosbattles.com:9877
p2p-peer-address = 34.226.76.22:9876
p2p-peer-address = mainnet.eosoasis.io:9876
p2p-peer-address = node.eosflare.io:1883
p2p-peer-address = p2p.eosio.cr:1976
p2p-peer-address = p2p.eosio.cr:5418
p2p-peer-address = eno.eosvan.io:19866
```

之后再执行 nodeos（这次不需要加后面的参数），这时便开始找邻近节点同步信息了。再执行 cleos get info，会出现图 4-12 所示的界面。

```
➜ eos git:(master) ✗ cleos get info
{
  "server_version": "5875549c",
  "chain_id": "aca376f206b8fc25a6ed44dbdc66547c36c6c33e3a119ffbeaef943642f0e906",
  "head_block_num": 1096,
  "last_irreversible_block_num": 1095,
  "last_irreversible_block_id": "000004478d691326c2363dc35c0ab247e2a862c2b827e6b627be0bfa14ca352b",
  "head_block_id": "00000448572c05b9b82d802fb3b154c1b04b12468d74c43a81f7c2276ec11e7d",
  "head_block_time": "2018-06-09T12:07:21.000",
  "head_block_producer": "eosio",
  "virtual_block_cpu_limit": 195293293,
  "virtual_block_net_limit": 3135124,
  "block_cpu_limit": 99999900,
  "block_net_limit": 1048576
}
```

图 4-12　连接主网信息

chain_id 确实是主网 ID，执行 cleos get block n（n 为区块号）也没有报错，连接主网成功！

仅供非商业用途或交流学习使用

2. get 下的 12 个二级子命令

链上各种数据和信息的查询主要由命令 get 来完成，get 下面有 12 个二级子命令，下面会逐一介绍该这 12 个命令的具体使用和信息查询方法。

1）info

功能：通过指定全节点 API 接口获取其连接的 EOS 区块链的基本信息。

命令 cleos get info 的用法如图 4-13 所示。

```
[cc@MBP] nodeos $ cleos get info
{
  "server_version": "980721a6",
  "chain_id": "aca376f206b8fc25a6ed44dbdc66547c36c6c33e3a119ffbeaef943642f0e906",
  "head_block_num": 7110721,
  "last_irreversible_block_num": 7110392,
  "last_irreversible_block_id": "006c7ef825a02c4743f227937786c5f1512ed3ff665a73715d210a66a5acd6e4",
  "head_block_id": "006c804125f306d8220dd425dc5083240defa7d2ec44d58832daef099e8f7b6c",
  "head_block_time": "2018-07-22T01:47:08.500",
  "head_block_producer": "eosriobrazil",
  "virtual_block_cpu_limit": 31164181,
  "virtual_block_net_limit": 1048576000,
  "block_cpu_limit": 199900,
  "block_net_limit": 1048576
}
```

图 4-13 cleos get info 示例

主网的 chain_id 为 aca376f206b8fc25a6ed44dbdc66547c36c6c33e3a119ffbeaef943642f0e906，通过 cleos get info 命令查询的 chain_id 为该字符串时，证明已经连接主网。

2）block

功能：从区块链中检索完整的区块信息，可根据指定区块号（block_num）或区块 ID 进行查询。

命令为 cleos get block [OPTIONS] block。

必填参数为 #block TEXT，用于表示要检索的块号或 ID。

非必填参数 [OPTIONS]，#-header-state 表示从可逆区块数据库中获取块头信息。

示例如下：

```
cleos get block 7830762
# 根据指定区块号查询块详情
cleos get block 00777cea85e75cde5f4fd861fc530d1fa8e46a3294549cf674976c648e34
    e11b
# 根据指定ID查询块详情
```

```
cleos get block 7830762 --header-state
# 根据指定可逆区块号查询块头信息
```

命令用法如图 4-14 所示。

```
[cc@MBP] nodeos $ cleos get block 7830762
{
  "timestamp": "2018-07-26T07:08:25.000",
  "producer": "eosisgravity",
  "confirmed": 0,
  "previous": "00777ce963dc4a04844ad2f8abedd412ba834a93060aa4ba67ca702a833cb58e",
  "transaction_mroot": "0000000000000000000000000000000000000000000000000000000000000000",
  "action_mroot": "12a12856a6525e760e0eedf282b5a1bb024ca3340fb94a1a382b4afda8118e97",
  "schedule_version": 216,
  "new_producers": null,
  "header_extensions": [],
  "producer_signature": "SIG_K1_K8rqyMBHsMPbuXwcx83dxFME5iHQvTBNCTRt92qYQ4QEAWSdNVHhZMSA1sXf6AkphSKJuSuh9oSiuG1ENgpRTXRTjpqqu1",
  "transactions": [],
  "block_extensions": [],
  "id": "00777cea85e75cde5f4fd861fc530d1fa8e46a3294549cf674976c648e34e11b",
  "block_num": 7830762,
  "ref_block_prefix": 1641566047
}
[cc@MBP] nodeos $ cleos get block 00777cea85e75cde5f4fd861fc530d1fa8e46a3294549cf674976c648e34e11b
{
  "timestamp": "2018-07-26T07:08:25.000",
  "producer": "eosisgravity",
  "confirmed": 0,
  "previous": "00777ce963dc4a04844ad2f8abedd412ba834a93060aa4ba67ca702a833cb58e",
  "transaction_mroot": "0000000000000000000000000000000000000000000000000000000000000000",
  "action_mroot": "12a12856a6525e760e0eedf282b5a1bb024ca3340fb94a1a382b4afda8118e97",
  "schedule_version": 216,
  "new_producers": null,
  "header_extensions": [],
  "producer_signature": "SIG_K1_K8rqyMBHsMPbuXwcx83dxFME5iHQvTBNCTRt92qYQ4QEAWSdNVHhZMSA1sXf6AkphSKJuSuh9oSiuG1ENgpRTXRTjpqqu1",
  "transactions": [],
  "block_extensions": [],
  "id": "00777cea85e75cde5f4fd861fc530d1fa8e46a3294549cf674976c648e34e11b",
  "block_num": 7830762,
  "ref_block_prefix": 1641566047
}
[cc@MBP] nodeos $ cleos get block 7830762 --header-state
{
  "id": "00777cea85e75cde5f4fd861fc530d1fa8e46a3294549cf674976c648e34e11b",
  "block_num": 7830762,
  "header": {
    "timestamp": "2018-07-26T07:08:25.000",
    "producer": "eosisgravity",
    "confirmed": 0,
    "previous": "00777ce963dc4a04844ad2f8abedd412ba834a93060aa4ba67ca702a833cb58e",
    "transaction_mroot": "0000000000000000000000000000000000000000000000000000000000000000",
    "action_mroot": "12a12856a6525e760e0eedf282b5a1bb024ca3340fb94a1a382b4afda8118e97",
    "schedule_version": 216,
    "header_extensions": [],
    "producer_signature": "SIG_K1_K8rqyMBHsMPbuXwcx83dxFME5iHQvTBNCTRt92qYQ4QEAWSdNVHhZMSA1sXf6AkphSKJuSuh9oSiuG1ENgpRTXRTjpqqu1"
  },
  "dpos_proposed_irreversible_blocknum": 7830605,
  "dpos_irreversible_blocknum": 7830441,
```

图 4-14　cleos get block 示例

3）account

功能：从区块链中检索指定账户详情。

命令为 cleos get account [OPTIONS] name。

必填参数 # name TEXT 表示要检索的账户名。

非必填参数 [OPTIONS]，# -j,--json 表示以 JSON 格式输出。

示例如下：

```
cleos get account paymesomeeos
# 查询账户paymesomeeos详情
cleos get account paymesomeeos -j
# 查询账户paymesomeeos详情并以JSON格式输出
```

命令用法如图 4-15 所示。

```
[[cc@MBP] nodeos $ cleos get account paymesomeeos
permissions:
     owner     1:    1 EOS7E4R7TRdsy1ERShxCMEUkLKekwjgQwzg9bL4o6aEoaxf98LjhR
        active     1:    1 EOS7E4R7TRdsy1ERShxCMEUkLKekwjgQwzg9bL4o6aEoaxf98LjhR
memory:
     quota:     3.557 KiB    used:     3.365 KiB

net bandwidth:
     staked:          0.0100 EOS           (total stake delegated from account to self)
     delegated:       0.0000 EOS           (total staked delegated to account from others)
     used:                 0 bytes
     available:        6.187 KiB
     limit:            6.187 KiB

cpu bandwidth:
     staked:          0.0100 EOS           (total stake delegated from account to self)
     delegated:       0.0000 EOS           (total staked delegated to account from others)
     used:                 0 us
     available:          102 us
     limit:              102 us

EOS balances:
     liquid:           4.0000 EOS
     staked:           0.0200 EOS
     unstaking:        0.0000 EOS
     total:            4.0200 EOS

producers:     <not voted>

[[cc@MBP] nodeos $ cleos get account paymesomeeos -j
{
  "account_name": "paymesomeeos",
  "head_block_num": 7833099,
  "head_block_time": "2018-07-26T07:28:14.500",
  "privileged": false,
  "last_code_update": "1970-01-01T00:00:00.000",
  "created": "2018-06-21T09:00:02.500",
  "core_liquid_balance": "4.0000 EOS",
  "ram_quota": 3642,
  "net_weight": 100,
  "cpu_weight": 100,
  "net_limit": {
    "used": 0,
    "available": 6335,
    "max": 6335
```

图 4-15 cleos get account 示例

4）code

功能：获取指定账户发布的智能合约代码信息。

命令为 cleos get code [OPTIONS] name。

必填参数 # name TEXT 表示要检索的账户名。

非必填参数 [OPTIONS]：

- # -c,--code TEXT 表示将智能合约保存为 .wast/wasm 格式并为文件命名。
- # -a,--abi TEXT 表示将智能合约保存为 .abi 格式并为文件命名。
- # --wasm 表示将智能合约保存为 wasm。

示例如下：

```
cleos get code eosio.Token
# 获取eosio.Token账户部署的合约code哈希值，未部署智能合约的账号返回结果为全0
cleos get code eosio.Token -c eosio.Token.code
# 获取eosio.Token账户部署的合约code哈希值，并将智能合约保存为名为eosio.Token.code的wast
  文件
cleos get code eosio.Token -a eosio.Token.abi
# 获取eosio.Token账户部署的合约code哈希值，并将智能合约保存为名为eosio.Token.abi的abi
  文件
cleos get code eosio.Token --wasm
# 获取eosio.Token账户部署的合约code哈希值，并将智能合约保存为wasm
```

命令用法如图 4-16 所示。

```
[cc@MBP] nodeos $ cleos get code eosio.token
code hash: 3e0cf4172ab025f9fff5f1db11ee8a34d44779492e1d668ae1dc2d129e865348
[cc@MBP] nodeos $ cleos get code eosio.token -c eosio.token.code
code hash: 3e0cf4172ab025f9fff5f1db11ee8a34d44779492e1d668ae1dc2d129e865348
saving wast to eosio.token.code
[cc@MBP] nodeos $ cleos get code eosio.token -a eosio.token.abi
code hash: 3e0cf4172ab025f9fff5f1db11ee8a34d44779492e1d668ae1dc2d129e865348
saving abi to eosio.token.abi
[cc@MBP] nodeos $ cleos get code eosio.token --wasm
code hash: 3e0cf4172ab025f9fff5f1db11ee8a34d44779492e1d668ae1dc2d129e865348
[cc@MBP] nodeos $
```

图 4-16　cleos get code 示例

5）abi

功能：返回指定账户的智能合约 abi 文件。

命令为 cleos get abi [OPTIONS] name。

必填参数 # name TEXT 表示要检索的账户名。

非必填参数 [OPTIONS]，# -f,--file TEXT 表示将返回结果保存在指定文件中。

示例如下：

```
cleos get abi eosio.Token
# 打印账户eosio.Token部署的合约的abi文件
cleos get abi eosio.Token -f Token.abi
# 将账户eosio.Token部署的合约的abi文件保存为Token.abi
```

命令用法如图 4-17 所示。

```
root@linqijun-ThinkPad-Edge-E531:/home/linqijun/eos/Docker# cleos get abi eosio.token
{
  "version": "eosio::abi/1.0",
  "types": [{
      "new_type_name": "account_name",
      "type": "name"
    }
  ],
  "structs": [{
      "name": "transfer",
      "base": "",
      "fields": [{
          "name": "from",
          "type": "account_name"
        },{
          "name": "to",
          "type": "account_name"
        },{
          "name": "quantity",
          "type": "asset"
        },{
          "name": "memo",
          "type": "string"
        }
      ]
    },{
      "name": "create",
      "base": "",
      "fields": [{
          "name": "issuer",
          "type": "account_name"
        },{
          "name": "maximum_supply",
          "type": "asset"
        }
      ]
    },{
```

图 4-17 cleos get abi 示例

6）table

功能：返回指定表中数据。

命令为 cleos get table [OPTIONS] contract scope table。

必填参数：

- # contract TEXT 表示待检索的表所在智能合约账户。
- # scope TEXT 表示待检索数据所在作用域。
- # table TEXT 表示待检索表名称。

非必填参数 [OPTIONS]：

- # -b,--binary UINT 表示返回二进制数据。
- # -l,--limit UINT 表示返回数据中的最大行数。
- # -k,--key TEXT 表示待请求数据在表中的键值。
- # -L,--lower TEXT 表示键的下限值的 JSON 表示，默认为 first。
- # -U,--upper TEXT 表示键的上限值的 JSON 表示，默认为 last。
- # --index TEXT 表示指定第几个索引，1 为第一个。
- # --key-type TEXT 表示 --index 的键值类型。

示例如下：

```
cleos get table eosio.Token user1 accounts
# 事先调用eosio.Token合约，为user1账户分发一些新建的Token，查询与合约对应的余额表数据
```

命令用法如图 4-18 所示。

```
root@linqijun-ThinkPad-Edge-E531:/home/linqijun/eos/Docker# cleos get table eosio.token user1 accounts
{
  "rows": [{
      "balance": "10000000.0000 LQJ"
    }
  ],
  "more": false
}
```

图 4-18　cleos get table 示例

7）currency

currency 下面又包含 balance 和 stats 两个三级子命令。

balance

功能：返回指定账户的某个 Token 的余额。

命令为 cleos get currency balance contract account [symbol]。

必填参数：

- # contract TEXT 表示 Token 发行合约账户名。
- # account TEXT 表示待检索账户名。

非必填参数 # symbol TEXT 表示待检索 Token 的名称，主网上默认 EOS。

非卖品，仅供非商业用途或交流学习使用

示例如下：

```
cleos get currency balance eosio.Token eosisgravity
# 获取账户eosisgravity的EOS余额
cleos get currency balance eosio.Token eosisgravity SYS
# 获取账户eosisgravity的SYS余额
```

命令用法如图 4-19 所示。

```
[cc@MBP] docker $ cleos get currency balance
ERROR: RequiredError: contract
Retrieve the balance of an account for a given currency
Usage: /opt/eosio/bin/cleos get currency balance contract account [symbol]

Positionals:
  contract TEXT               The contract that operates the currency (required)
  account TEXT                The account to query balances for (required)
  symbol TEXT                 The symbol for the currency if the contract operates multiple currencies

[cc@MBP] docker $ cleos get currency balance eosio.token eosisgravity
13088.7400 EOS
[cc@MBP] docker $ cleos get currency balance eosio.token eosisgravity SYS
[cc@MBP] docker $
```

图 4-19 cleos get currency balance 示例

stats

功能：返回指定 Token 的发行状态（初始总额、发行人等）。

命令为 cleos get currency stats contract symbol。

必填参数：

- # contract TEXT 表示 Token 发行合约账户名。
- # symbol TEXT 表示待检索 Token 的名称。

示例如下：

```
cleos get currency stats eosio.Token EOS
# 获取主网上EOS的发行情况
```

命令用法如图 4-20 所示。

```
[cc@MBP] docker $ cleos get currency stats eosio.token EOS
{
  "EOS": {
    "supply": "1006416238.7968 EOS",
    "max_supply": "10000000000.0000 EOS",
    "issuer": "eosio"
  }
}
[cc@MBP] docker $
```

图 4-20 cleos get currency stats 示例

非卖品，仅供非商业用途或交流学习使用

8）accounts

功能：返回指定公钥所控制的所有账户。

命令为 cleos get accounts publice_key。

必填参数 # public_key TEXT 表示待检索的 EOS 公钥。

示例如下：

```
cleos get accounts EOS1111111111111111111111111111111114T1Anm
# 查看公钥EOS1…Anm所控制账户
```

命令用法如图 4-21 所示。

```
[cc@MBP] docker $ cleos get accounts EOS1111111111111111111111111111111114T1Anm
{
  "account_names": [
    "genesisblock",
    "octblackhole",
    "uipblackhole"
  ]
}
[cc@MBP] docker $
```

图 4-21　cleos get accounts 示例

9）servants

功能：返回指定账户所控制的所有账户。

命令为 cleos get servants account。

必填参数 # account TEXT 表示待检索账户名。

示例如下：

```
cleos get servants eosio
# 查看eosio账户所控制的账户
```

命令用法如图 4-22 所示。

10）transaction

功能：在区块链上检索某个交易的详细信息。

命令为 cleos get transaction [OPTIONS] id。

必填参数 # id TEXT 表示待检索的交易 ID。

非必填参数 [OPTIONS]，# -b,--block-hint UINT 表示该交易可能存在的区块号。

示例如下：

非卖品，仅供非商业用途或交流学习使用


```
[cc@MBP] docker $ cleos get servants eosio
{
  "controlled_accounts": [
    "eoshappyvote",
    "eosio.bpay",
    "eosio.msig",
    "eosio.names",
    "eosio.ram",
    "eosio.ramfee",
    "eosio.saving",
    "eosio.stake",
    "eosio.token",
    "eosio.unregd",
    "eosio.vpay",
    "eosvrtokenss",
    "gm2dimrqgige",
    "nvoyiooiyovn",
    "stoponnopots",
    "testtodelete",
    "toohottohoot"
  ]
}
```

图 4-22 cleos get servants 示例

```
cleos get transaction f3fc34dd9094c828e951d7d433f6f621c41c3133a58da0e6899814277a7f7ce0
```

命令用法如图 4-23 所示。

```
[cc@MBP] docker $ cleos get transaction f3fc34dd9094c828e951d7d433f6f621c41c3133a58da0e6899814277a7f7ce0
{
  "id": "f3fc34dd9094c828e951d7d433f6f621c41c3133a58da0e6899814277a7f7ce0",
  "trx": {
    "receipt": {
      "status": "executed",
      "cpu_usage_us": 1254,
      "net_usage_words": 17,
      "trx": [
        1,{
          "signatures": [
            "SIG_K1_KkaK9JERBiHZQZyThDpQJagnPg9ihz5RH4TvrwyyBtfx5SPgny6wUsRtDS3Rx6cvX8DS3JtsUbkDpswjUycduU
          ],
          "compression": "none",
          "packed_context_free_data": "",
          "packed_trx": "00c65a5bb110c5d8233600000000000100a6823403ea3055000000572d3ccdcd013086102104365 5fb00
100000000000004454f530000000063132393434350 0"
        }
      ]
```

图 4-23 cleos get transaction 示例

11）actions

功能：在区块链上检索某个账户的所有 actions。

命令为 cleos get actions [OPTIONS] account_name [PoS] [offset]。

必填参数 # account_name TEXT 表示待检索账户名。

非卖品，仅供非商业用途或交流学习使用。

非必填参数 [OPTIONS]：

- # PoS INT 表示 action 的序号，默认 -1，即最后一个。
- # offset INT 表示 action 的数量，为正则向后数 offset 个 action，负则向前数。
- # -j,--json 表示以 JSON 格式打印。
- # --full 表示打印不缩短 action 的 JSON 格式。
- # --pretty 表示打印优化后的 JSON 格式。
- # --console 表示打印 action 在控制台的输出结果。

示例如下：

```
cleos get actions eosisgravity
cleos get actions eosisgravity 101 0
cleos get actions eosisgravity 101 0 --full
cleos get actions eosisgravity 101 0 --pretty
cleos get actions eosisgravity 101 0 --console
```

命令用法如图 4-24 所示。

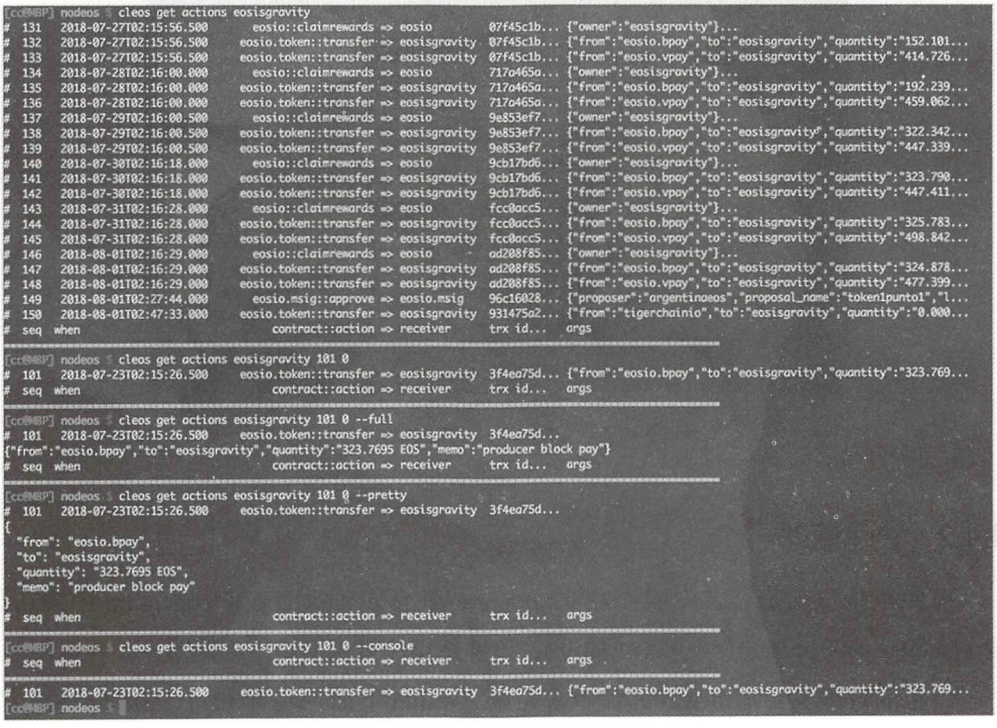


图 4-24　cleos get actions 示例

非卖品，仅供非商业用途或交流学习使用

12）schedule

功能：检索 EOS 区块链当前的节点出块顺序安排。

命令为 cleos get schedule [OPTIONS]。

非必填参数 [OPTIONS]，# -j,--json 表示以 JSON 格式打印检索结果。

示例如下：

```
cleos get schedule
cleos get schedule -j
```

命令用法如图 4-25 所示。

```
[cc@MBP] nodeos $ cleos get schedule
active schedule version 252
    Producer       Producer key
    ============== ==================
    argentinaeos   EOS7n4UUEDQRWeJ5UmCf9yqWXY5fsTtbo78HyYa5u8bM1xwa5DwRj
    bitfinexeos1   EOS6sgKjHUFtY1XxxQaMDwfxBac6nDBibVzZHb8LFMVmvSjcCdDhE
    eos42freedom   EOS4tw7vH62TcVtMgm2tjXzn9hTuHEBbGPUK2eos42ssY7ip4LTzu
    eosamsterdam   EOS8HDwgQnKMHAxzva9eyHsPeETv23JSQ6RRzQjJLvCsZ6b1ySgQo
    eosasia11111   EOS76gG6ATpqfVf5KrVjh3f4JAa4EKzAwWabTucNQ4Xv2TmVAj9bN
    eosauthority   EOS4va3CTmAcAAXsT26T3EBWqYHgQLshyxsozYRgxWm9tjmy17pVV
    eosbeijingbp   EOS5dGpcEhwB4VEhhXEcqtZs9EQj5HeetuXDnsAGVHMXHAFdMjbdj
    eosbixinboot   EOS7QC1XfAtkYeLjbHQjcDWVqUsxuSJ3YRhNyG93VAv2u3uHopGVp
    eoscanadacom   EOS5HYV7rWeRxpZMCooe8YHRFQHKK7ncdmmUMTe3wCMaY2EvyVzUx
    eosdacserver   EOS6VkageCqyz1X97n7E1JyjPoZu8HEX6CVujY5A8zk3fPyV3vzZK
    eoshuobipool   EOS5XKswW26cR5VQeDGwgNb5aixv1AMcKkdDNrC59KzNSBfnH6TR7
    eosisgravity   EOS55HTTjoxVX1zVpW8pabxyg8b1J3SEnG5D8D3y3KgrnSbLpELfE
    eoslaomaocom   EOS8QgURqo875qu3a8vgZ58qBeu2cTehe9zAWRfpdCXAQipicu1Fi
    eosnewyorkio   EOS6GVX8eUqC1gN1293B3ivCNbifbr1BT6gzTFaQBXzWH9QNKVM4X
    eosriobrazil   EOS7RioGoHQnhv2fJEiciP9Q7J8JgfJYFcyofVkmCqMop8Q1PzgqP
    eosswedenorg   EOS7SGSBsWhSob6TEric6u3TGodcc1uXFcqSrquJ3Etuqcbb3VnNY
    helloeoscnbp   EOS5CUnzSK6bFUQcteYom8m1HivZFw4LggbmhmigxNuQyHhQhoxR4
    libertyblock   EOS7TqVs7LpMzJwikiWkGTG9UGSwcQVhxP2ZKBM7FCbUwcrryNgHP
    starteosiobp   EOS4wZZXm994byKANLuwHD6tV3R3Mu3ktc41aSVXCBaGnXJZJ4pwF
    teamgreymass   EOS4xurP5oZk4WvMUp9neKkw94DJ2iq48GtMD5Dff3GKe6krh2ctR
    zbeosbp11111   EOS7rhgVPWWyfMqjSbNdndtCK8Gkza3xnDbUupsPLMZ6gjfQ4nX81

pending schedule empty

proposed schedule empty
```

图 4-25 cleos get schedule 示例

4.4.2 钱包管理

要使用钱包功能，首先需要在 local/share/eosio/nodeos/config/config.ini 中确认加载了 wallet_api_plugin（钱包 API 插件），该详细操作可以回顾前面几节。

钱包的管理主要是由 wallet 命令来完成。wallet 下面一共有 12 个二级子命令，下面具体介绍这 12 个命令是如何做钱包管理的。

非卖品，仅供非商业用途或交流学习使用

1. create

功能：创建一个新的钱包。

命令为 cleos wallet create [OPTIONS]。

非必填参数 [OPTIONS]，# -n,--name TEXT=default 表示创建钱包名，如果缺省则默认创建一个名为 default 的钱包。

示例如下：

```
cleos wallet create -n test1
# 创建一个名为test1的钱包，并且输出该钱包的密码
```

命令用法如图 4-26 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet create -n test1
"/usr/local/eosio/bin/keosd" launched
Creating wallet: test1
Save password to use in the future to unlock this wallet.
Without password imported keys will not be retrievable.
"PW5JHmp8fpX1AzZTqGdrPNBynuaVernu1N7PhgshsFakyaAgruG9h"
```

图 4-26　cleos wallet create 示例

创建了一个名为 test1 的钱包，该钱包密码为 PW5JaZitmdV4nUcumR7Z1rE6DG1GC8E9bMs94aAsQABsgGpxGb1SK。创建的钱包在系统主目录下的 eosio-wallet 文件夹中，可以打开文件查看钱包信息如下（该钱包信息是一个简单的 JSON 对象，只有一个“cipher_keys”）：

```
{
    "cipher_keys": "e0d2d71b744f0b5287018d4d4a867b957d8cc1a27302ea995578240c
        08981b3733d9730e158495bce33b71902351018cf6bf41275efc6a96d035890401d3
        688fd1c88bc5eb1382c86112012381b5c0cdeb3100ac53a15fb87072176bb87366cd
        364b58e03c2f96e1bf6fba4b069561cf07dca1386dbc7d30d950f87bc9975dc55504
        1e262b025a30d02fb94884e2110e"
}
```

2. open

功能：打开一个处于关闭状态的钱包。

命令为 cleos wallet open [OPTIONS]。

非必填参数 [OPTIONS]，# -n,--name TEXT 表示要打开的钱包名。

示例如下：

非卖品，仅供非商业用途或交流学习使用


```
cleos wallet open -n test1
# 打开test1钱包
```

命令用法如图 4-27 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet open -n test1
Opened: test1
guo@guo-XPS-13-9360:~$
```

图 4-27 cleos wallet open 示例

3. lock

功能：锁定钱包（未使用的钱包 15 分钟后会自动上锁）。

命令为 cleos wallet lock [OPTIONS]。

非必填参数 [OPTIONS]，# -n,--name TEXT 表示要锁定的钱包名。

示例如下：

```
cleos wallet lock -n test1
#锁定钱包test1
```

命令用法如图 4-28 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet lock -n test1
Locked: test1
guo@guo-XPS-13-9360:~$
```

图 4-28 cleos wallet lock 示例

4. lock_all

功能：锁定所有未上锁的钱包。

示例如下：

```
cleos wallet lock_all
```

命令用法如图 4-29 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet lock_all
Locked All Wallets
guo@guo-XPS-13-9360:~$
```

图 4-29 cleos wallet lock_all 示例

5. unlock

功能：解锁一个上锁的钱包。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

命令为 cleos wallet unlock [OPTIONS]。

非必填参数 [OPTIONS]：

- # -n,--name TEXT 表示需要解锁的钱包名。
- #--password TEXT 表示需要解锁的钱包密码。

示例如下：

```
cleos wallet unlock -n test1
#解锁钱包test1
```

命令用法如图 4-30 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet unlock -n test1
password: Unlocked: test1
guo@guo-XPS-13-9360:~$
```

图 4-30　cleos wallet unlock 示例

6. import

功能：给指定的钱包导入一个私钥。

命令为 cleos wallet import [OPTIONS]。

非必填参数 [OPTIONS]：

- # -n,--name TEXT 表示钱包名。
- #--private-key TEXT 表示导入钱包的私钥。

示例如下：

```
cleos wallet import -n test1 --private-key 5Ken8D5HY82UhZaB9Y4E33tn4odNmnAe42PkbdBp
ZwCMZ3BGXjj
#钱包test1导入私钥5Ken...BGXjj
```

命令用法如图 4-31 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet import -n test1 --private-key 5Ken8D5HY82UhZaB9Y4E33tn
4odNmnAe42PkbdBpZwCMZ3BGXjj
imported private key for: EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati
guo@guo-XPS-13-9360:~$
```

图 4-31　cleos wallet import 示例

7. remove_key

功能：对指定的钱包移除一个密钥对。

命令为 cleos wallet remove_key [OPTIONS] key。

仅供非商业用途或交流学习使用

必填参数 #key TEXT 表示所要移除密钥对中的公钥。

非必填参数 [OPTIONS]：

- # -n,--name TEXT 表示要移除密钥对的钱包名。
- #--password TEXT 表示钱包密码。

示例如下：

```
cleos wallet remove_key -n test1 EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati
# 从钱包test1中移除与公钥EOS53...Wati对应的密钥对
```

命令用法如图 4-32 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet remove_key -n test1 EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati
password: removed private key for: EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati
guo@guo-XPS-13-9360:~$
```

图 4-32 cleos wallet remove_key 示例

8. create_key

功能：在钱包内创建密钥对。

命令为 cleos wallet create_key [OPTIONS] [key_type]。

非必填参数 [key_type]，#key_type K1/R1 表示私钥对应的类型，默认为基于 K1 曲线生成密钥对。

非必填参数 [OPTIONS]，# -n,--name TEXT=default 表示钱包名。

示例如下：

```
cleos wallet create_key -n test1 K1
#在钱包test1中创建基于K1曲线生成的密钥对
cleos wallet create_key -n test1 R1
#在钱包test1中创建基于R1曲线生成的密钥对
```

命令用法如图 4-33 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet create_key -n test1 K1
Created new private key with a public key of: "EOS5nYCCmPeuUv66G25Cx7JVStjcMuwvRfpzVeX2eZc6vKatRJWPt"
guo@guo-XPS-13-9360:~$ cleos wallet create_key -n test1 R1
Created new private key with a public key of: "PUB_R1_834f38Vf6a8sdu4N9JaRM5Bzc7v3wWVys9fo9T4VxoAbv8NJHp"
guo@guo-XPS-13-9360:~$
```

图 4-33 cleos wallet create_key 示例

仅供非商业用途或交流学习使用

9. list

功能：列出当前所有已打开打钱包，钱包名带“*”表示未上锁。

命令为 cleos wallet list。

命令用法如图 4-34 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet list
Wallets:
[
  "eosio",
  "test1 *",
  "test3"
]
guo@guo-XPS-13-9360:~$
```

图 4-34　cleos wallet list 示例

10. keys

功能：列出当前所有未上锁钱包中的所有与密钥对对应的私钥。

命令为 cleos wallet keys。

命令用法如图 4-35 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet keys
[
  "EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati",
  "EOS8WMS8mscxA5xTAyRihp8vG4qkR8PeQcspSbW7dwsBKVoKmZs9Y"
]
guo@guo-XPS-13-9360:~$
```

图 4-35　cleos wallet keys 示例

11. private_keys

功能：列出指定钱包中的所有密钥对。

命令为 cleos wallet private_keys [OPTIONS]。

非必填参数 [OPTIONS]：

- # -n,--name TEXT=default 表示钱包名。
- #--password TEXT 表示钱包密码。

示例如下：

```
cleos wallet private_keys -n test1
#列出钱包test1中所有的密钥对
```

命令用法如图 4-36 所示。

仅供非商业用途或交流学习使用

```
guo@guo-XPS-13-9360:~$ cleos wallet private_keys -n test1
password: [[
    "EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati",
    "5Ken8D5HY82UhZaB9Y4E33tn4odNmnAe42PkbdBpZwCMZ3BGXjj"
  ],[
    "EOS8WMS8mscxA5xTAyRihp8vG4qkR8PeQcspSbW7dwsBKVoKmZs9Y",
    "5HrrpTtaZGpCwsvB74Fcg1HkAKAQHFZBGUsXHefXtD7rHoFRgiX"
  ]
]
guo@guo-XPS-13-9360:~$
```

图 4-36 cleos wallet private_keys 示例

12. stop

功能：关闭 kesod 程序（所有钱包会被关闭，如需再使用某个钱包，需用命令 open 打开）。

命令为 cleos wallet stop。

命令用法如图 4-37 所示。

```
guo@guo-XPS-13-9360:~$ cleos wallet stop
OK
guo@guo-XPS-13-9360:~$ cleos wallet list
"/usr/local/eosio/bin/keosd" launched
Wallets:
[]
guo@guo-XPS-13-9360:~$
```

图 4-37 cleos wallet stop 示例

4.4.3 账户创建与权限管理

第 2 章中详细讲解了账户的权限管理，本节根据上述内容介绍用命令行方式在本地测试网上进行账户的创建和权限管理。

账户创建可以使用命令 cleos create account [OPTIONS] creator name OwnerKey [ActiveKey] 实现，其中 creator 是创建者，一般会使用系统原生账户“ eosio”；name 为要创建的账户名。OwnerKey 和 ActiveKey 是分别控制该账户 owner 权限和 active 权限的密钥对中的公钥，两者也可以是同一个公钥，此时表明该账户的 owner 权限和 active 权限由同一个密钥控制。另外需要注意的是，这两个密钥对都需要关联钱包，并且创建账户时要需解锁钱包。

在启动本地测试网之后，使用上述命令创建账户 sg，由同一个密钥对控制 sg 账户的 owner 权限和 active 权限。账户创建之后，我们可以使用命令 cleos get accounts public-key 通过公钥查询到拥有相应控制权限的账户，命令用法如图 4-38 所示。

仅供非商业用途或交流学习使用

```
guo@guo-XPS-13-9360:~/eos/build/programs/cleos$ cleos create account eosio sg EOS71fmsHe
EwG9EiYsZDRhGZScavyG5UgmedJCiWG7edajhEzzyHM EOS71fmsHeEwG9EiYsZDRhGZScavyG5UgmedJCiWG7ed
ajhEzzyHM
executed transaction: 81e5e1778e31af7717fcaba9f84737ef89466e4e94fa357b8b8a40a641ae7c7b
200 bytes  25465 us
#         eosio <= eosio::newaccount          {"creator":"eosio","name":"sg","owner":{
"threshold":1,"keys":[{"key":"EOS71fmsHeEwG9EiYsZDRhGZScavyG...
warning: transaction executed locally, but may not be confirmed by the network yet    ]
guo@guo-XPS-13-9360:~/eos/build/programs/cleos$
```

图 4-38　cleos get accounts 示例

链上已存在的账户也可以使用命令 cleos get account name 查询账户的详细信息，其中包含账户的权限控制信息、账户拥有的系统资源信息。

```
cleos get account sg
permissions:
owner     1:    1 EOS71fmsHeEwG9EiYsZDRhGZScavyG5UgmedJCiWG7edajhEzzyHM
active     1:    1 EOS71fmsHeEwG9EiYsZDRhGZScavyG5UgmedJCiWG7edajhEzzyHM
memory:
    quota:       unlimited  used:      2.66 KiB

net bandwidth:
    used:                   unlimited
    available:              unlimited
    limit:                  unlimited

cpu bandwidth:
    used:                   unlimited
    available:              unlimited
    limit:                  unlimited
```

sg 账户的 owner 权限和 active 权限由同一个密钥控制，我们也可以通过命令行更改权限的密钥。我们通过以下命令更改 active 权限的密钥：

```
cleos set account permission sg  active '{"threshold": 1, "keys": [{"key":"
    EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati", "weight": 1}],
    "accounts": []}' owner
```

其中：

- threshold 表示权限阈值，只有等于阈值时才能获取该权限，否则授权会失败。
- keys 是该权限已授权的密匙：keys 中的 key 为授权的密匙对的公匙，weight 为授权密匙对的权重，如果授权的权重小于阈值，则无法完成授权。
- accounts 是该权限已授权的账户：permission 是被授权用户的权限，这里指 sg 账户的 active 权限，最后是 owner 权限签名，只有 owner 权限才能改变用户的权限。

仅供非商业用途或交流学习使用

经过上述命令过后，查看账户 sg 的详细信息可得以下信息，由信息可知，active 权限密钥更改成功。

```
$cleos get account sg
permissions:
    owner     1:    1 EOS71fmsHeEwG9EiYsZDRhGZScavyG5UgmedJCiWG7edajhEzzyHM
    active     1:    1 EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati
memory:
    quota:       unlimited  used:     32.51 KiB

net bandwidth:
    used:               unlimited
    available:          unlimited
    limit:              unlimited

cpu bandwidth:
    used:               unlimited
    available:          unlimited
    limit:              unlimited
```

我们还可以自己定义其他权限，自定义权限是通过 active 签名绑定的权限。例如可以使用下列命令给 sg 账户自定义 vote 权限：

```
cleos set account permission sg vote'{"threshold":1,"keys":[{"key":"EOS62k
    QCZvk6iasLaidZuE6nJocZFdvVWeQuFksUhMoyf8n3zoc2W","weight":1}],"accoun
    ts":[]}'
```

active 给 sg 账户自定义 vote 权限，查看账户可以看到有 vote 权限：

```
$ cleos get account sg
permissions:
    owner     1:    1 EOS71fmsHeEwG9EiYsZDRhGZScavyG5UgmedJCiWG7edajhEzzyHM
    active     1:    1 EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati
    vote     1:    1 EOS62kQCZvk6iasLaidZuE6nJocZFdvVWeQuFksUhMoyf8n3zoc2W
memory:
    quota:       unlimited  used:     32.84 KiB

net bandwidth:
    used:               unlimited
    available:          unlimited
    limit:              unlimited

cpu bandwidth:
    used:               unlimited
```

仅供非商业用途或交流学习使用

```
    available:          unlimited
    limit:              unlimited
```

对于自定义的权限，我们也可以使用：cleos set account permission name_account authority_name 'NULL' active 命令删除，如图 4-39 所示。

```
guo@guo-XPS-13-9360:~/eos/build/contracts$ cleos get account sg
permissions:
     owner     1:    1 EOS71fmsHeEwG9EiYsZDRhGZScavyG5UgmedJCiWG7edajhEzzyHM
        active     1:    1 EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati
           vote     1:    1 EOS62kQCZvk6iasLaidZuE6nJocZFdvVWeQuFksUhMoyf8n3zoc2W
memory:
     quota:       unlimited  used:     32.84 KiB

net bandwidth:
     used:               unlimited
     available:          unlimited
     limit:              unlimited

cpu bandwidth:
     used:               unlimited
     available:          unlimited
     limit:              unlimited


guo@guo-XPS-13-9360:~/eos/build/contracts$ cleos set account permission sg vote 'NULL' active
executed transaction: 7115eb9931f2497b31bab6fc293b03cf8f83db26fe60849a37e8b4b0d19043c0  112 byte
s  253 us
#         eosio <= eosio::deleteauth            {"account":"sg","permission":"vote"}
warning: transaction executed locally, but may not be confirmed by the network yet    ]
guo@guo-XPS-13-9360:~/eos/build/contracts$ cleos get account sg
permissions:
     owner     1:    1 EOS71fmsHeEwG9EiYsZDRhGZScavyG5UgmedJCiWG7edajhEzzyHM
        active     1:    1 EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati
memory:
     quota:       unlimited  used:     32.51 KiB

net bandwidth:
     used:               unlimited
     available:          unlimited
     limit:              unlimited

cpu bandwidth:
     used:               unlimited
     available:          unlimited
     limit:              unlimited
```

图 4-39　cleos set accounts 示例

4.4.4　代币的交易

EOS 系统中有自带的合约 eosio.Token，该合约用于代币发行。因此在发行自己的代币之前，我们先部署该合约和创建自己发币账户和测试账户。

创建 eosio.Token 和 test 账户，如图 4-40 所示。

非卖品，仅供非商业用途或交流学习使用


```
guo@guo-XPS-13-9360:~$ cleos create account eosio eosio.token EOS6v65fupFdgC4zGZRrvo9TcbTnmGkXApmq
1wtYxpUmHgwVEzZfj EOS6v65fupFdgC4zGZRrvo9TcbTnmGkXApmq1wtYxpUmHgwVEzZfj
executed transaction: 27d2dcbf9b078a6b7f9b696fabf7210b68146efa0d34c8d60c41e8d2280b9273  200 bytes
 827 us
#         eosio <= eosio::newaccount            {"creator":"eosio","name":"eosio.token","owner":{"
threshold":1,"keys":[{"key":"EOS6v65fupFdgC4zGZRrv...
2018-08-21T07:42:49.458 thread-0   main.cpp:432                 print_result   warning: transacti
on executed locally, but may not be confirmed by the network yet
guo@guo-XPS-13-9360:~$ cleos create account eosio test EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMr
ZT8CchWati EOS53BLGGa9adoq5PoFt7p5Q4JEpypjSLJjJ64bziMrZT8CchWati
executed transaction: 0016bce2b7793e551f68e1df45c95d205e38429b68e72b3d3ac088946d6e91ed  200 bytes
 586 us
#         eosio <= eosio::newaccount            {"creator":"eosio","name":"test","owner":{"thresho
ld":1,"keys":[{"key":"EOS53BLGGa9adoq5PoFt7p5Q4JEp...
warning: transaction executed locally, but may not be confirmed by the network yet    ]
```

图 4-40 创建账户

进入 eos/build/contracts 目录下部署智能合约 eosio.Token：

```
cleos set contract eosio.Token eosio.Token/
```

关于智能合约部署和调用的详细介绍见 4.5 节。

部署完该智能合约后，就可以使用账户发行代币了。在这里我们使用 eosio.Token 账户发行 10000000000 个 TEST 代币，命令如下：

```
cleos push action eosio.Token create '["eosio.Token","10000000000.0000
    TEST","create"]' -p eosio.Token。
```

创建代币之后，我们还需要把代币分配给账户，使用以下命令把代币全部分配给 eosio.Token 账户：

```
cleos push action eosio.Token issue '["eosio.Token", "10000000000.0000
    TEST","issue"]' -p eosio.Token
```

分配完成之后，eosio.Token 账户里会有 10000000000 个 TEST 代币，我们可以使用“cleos get currency balance eosio.Token+ 账户名”来查询该账户下的代币数量。

通过转账命令 transfer 可以把代币转给指定的账户，我们可以分别使用以下两个命令从 eosio.Token 账户转 100000000 个 TEST 给 test 账户：

```
cleos push action eosio.Token transfer '["eosio.Token","test","100000000.000
0RMB","transfer"]' -p eosio.Token
cleos transfer eosio.Token test "100000000.0000 TEST"
```

完整的转账命令示意如图 4-41 所示。

由图可知，eosio.Token 账户和 test 账户初始的代币余额分别是 10000000000.0000 TEST、0TEST，通过转账命令转账之后，eosio.Token 账户和 test 账户的代币余额分别

非卖品，仅供非商业用途或交流学习使用

是 9900000000.0000 TEST、100000000TEST。由此可知，代币交易转账成功。

```
guo@guo-XPS-13-9360:~$ cleos push action eosio.token create '["eosio.token","10000000000.0000 TEST","creat
e"]' -p eosio.token
executed transaction: 381cdcb17ce14e3af1411aeeb8b82391e78a648af90640d1064f3abe7f35e77b  120 bytes  349 us
#   eosio.token <= eosio.token::create         {"issuer":"eosio.token","maximum_supply":"10000000000.0000
 TEST"}
warning: transaction executed locally, but may not be confirmed by the network yet    ]
guo@guo-XPS-13-9360:~$ cleos push action eosio.token issue '["eosio.token", "10000000000.0000 TEST","issue
"]' -p eosio.token
executed transaction: adfff7dac56c53a81b3e059247a08edc85fe540ceecbd21fe6f3bef15ded3fcc  128 bytes  432 us
#   eosio.token <= eosio.token::issue          {"to":"eosio.token","quantity":"10000000000.0000 TEST","me
mo":"issue"}
warning: transaction executed locally, but may not be confirmed by the network yet    ]
guo@guo-XPS-13-9360:~$ cleos push action eosio.token transfer '["eosio.token","test","100000000.0000TEST",
"transfer"]' -p eosio.token
2018-10-24T14:17:15.405 thread-0   asset.cpp:79                 from_string          ] 3010011 asset_type
_exception: Invalid asset
Asset's amount and symbol should be separated with space
    {}
    thread-0  asset.cpp:42 from_string
2018-10-24T14:17:15.405 thread-0   asset.cpp:79                 from_string          ] from: 100000000.00
00TEST
Error 3010011: Invalid asset
Error Details:
Asset's amount and symbol should be separated with space
rethrow
guo@guo-XPS-13-9360:~$ cleos push action eosio.token transfer '["eosio.token","test","100000000.0000 TEST"
,"transfer"]' -p eosio.token
executed transaction: 09eece1b704dea916dd7ff98469f407d93ceaff53d5508385c2a12356b1a22f6  136 bytes  445 us
#   eosio.token <= eosio.token::transfer       {"from":"eosio.token","to":"test","quantity":"100000000.00
00 TEST","memo":"transfer"}
#         test <= eosio.token::transfer        {"from":"eosio.token","to":"test","quantity":"100000000.00
00 TEST","memo":"transfer"}
warning: transaction executed locally, but may not be confirmed by the network yet    ]
guo@guo-XPS-13-9360:~$ cleos get currency balance eosio.token test
100000000.0000 TEST
guo@guo-XPS-13-9360:~$ cleos get currency balance eosio.token eosio.token
9900000000.0000 TEST
guo@guo-XPS-13-9360:~$
```

图 4-41　转账过程

4.4.5　交易的签名与发送

EOS 系统中事务的签名与发送主要是由 sign 命令来完成的，其用法如下：

```
cleos sign [OPTIONS] transaction
```

非必填参数 [OPTIONS]，#transaction 表示需要签名的事务。

非必填参数 [OPTIONS]：

- #-k,--private-key TEXT 表示事务签名需要的私钥。
- #-c,--chain-id TEXT 表示事务签名所在的网络标识符。
- #-p,--push-transaction 表示事务签名后推送该事务。

示例如下：

非卖品，仅供非商业用途或交流学习使用


```
cleos sign '{"ref_block_prefix":1}'
```

```
$ cleos sign '{"ref_block_prefix":1}'
2018-08-23T07:34:24.806 thread-0    main.cpp:2434       operator()] grabbing
    chain_id from nodeos
private key: {
    "expiration": "1970-01-01T00:00:00",
    "ref_block_num": 0,
    "ref_block_prefix": 1,
    "max_net_usage_words": 0,
    "max_cpu_usage_ms": 0,
    "delay_sec": 0,
    "context_free_actions": [],
    "actions": [],
    "transaction_extensions": [],
    "signatures": [
        "SIG_K1_KdFCRFM9U32BbZoohBztX9STU66e3zEwos4j7qubje6fXrEhPtYND9xLQ6e1
            hncphdC96j3rpp4Jco2SLppesX9TBiQeLd"
    ],
    "context_free_data": []
}
```

4.4.6 系统资源获取

EOS 系统中的资源主要有 CPU、net bandwidth 和 RAM 三种，本节主要介绍如何通过命令行获取以上三种资源。

1.CPU 和 Bandwidth

EOS 系统内抵押 Token 获取 CPU 与 Bandwidth，赎回抵押 Token 分别由命令 cleos system delegatebw 和 undelegatebw 完成，用法如下。

抵押 Token 获取 CPU 与 Bandwidth 的命令：

```
cleos system delegatebw [OPTIONS] from receiver stake_net_quantity stake_
    cpu_quantity
```

必填参数：

- #from TEXT 表示抵押的账户。
- #receiver TEXT 表示收取抵押 EOS 的账户。
- #stake_net_quantity TEXT 表示为获取 NET 所抵押的 EOS 数量。
- #stake_cpu_quantity TEXT 表示为获取 CPU 所抵押的 EOS 数量。

非卖品，仅供非商业用途或交流学习使用

非必填参数 [OPTIONS]：

- #-h,--help 表示打印帮助信息并退出。
- #--buyram TEXT 表示购买 RAM 数量。
- #--transfer 表示转移投票权。
- #-x,--expiration 表示交易过期时间，默认为 30s。
- #-f,--force-unique 表示使得交易唯一。
- #-s,--skip-sign 表示跳过签名。
- #-j,--json 表示以 JSON 格式打印。
- #-d,--dont-broadcast 表示不向全网广播。
- #-r,--ref-block TEXT 表示设置 TAPoS 使用的区块高度或区块 id。
- #-p,--permission TEXT ... 表示对该交易进行授权的账户及权限。
- #--max-cpu-usage-ms UINT 表示设置 CPU 使用量上限，默认无限。
- #--max-net-usage UINT 表示设置 NET 使用量上限，默认无限。

赎回抵押的 Token 命令为：

```
cleos system undelegatebw [OPTIONS] from receiver unstake_net_quantity
    unstake_cpu_quantity
```

必填参数

- #from TEXT 表示抵押的 EOS 赎回的账户名。
- #receiver TEXT 表示抵押的 EOS 被收回账户名。
- #unstake_net_quantity TEXT 表示赎回 Bandwidth 所抵押的 EOS 数量。
- #unstake_cpu_quantity TEXT 表示赎回 CPU 所抵押的 EOS 数量。

非必填参数 [OPTIONS]，同 delegatebw。

2.RAM

EOS 系统内购买出售 RAM 由命令 cleos system buyram/sellram 完成。

购买 RAM 命令为：

```
cleos system buyram [OPTIONS] payer receiver amount
```

必填参数：

- # payer TEXT 表示购买 RAM 付款账户。

非卖品，仅供非商业用途或交流学习使用

- # receiver TEXT 表示接收购买的 RAM 账户。
- # amount TEXT 表示购买 RAM 花费的 EOS 数量或者 RAM kb 数。

非必填参数 [OPTIONS]，同 delegatebw。

出售 RAM 命令为：

```
cleos system sellram [OPTIONS] account bytes
```

必填参数：

- # account TEXT 表示出售 ram 获取 EOS 的账户。
- # bytes UINT 表示出售 ram 的 bytes 量。

非必填参数 [OPTIONS]，同 delegatebw。

4.5 智能合约的部署与调用

EOS 系统内有自带智能合约，例如 eosio.bios、eosio.msig、eosio.Token、exchange 等。编译后的系统工程的智能合约都在 eos/build/constracts 文件夹下。

智能合约需要由一个账号来部署，并且一个账号只能部署一个智能合约，在部署合约之前需要确认账户公钥所绑定的钱包处于解锁状态。现在使用上一节创建的账户 sg 部署智能合约 helloeos，其合约具体内容是“ Welcome to EOS! ”，在 eos/build/constracts 目录下使用命令 cleos set contract sg ./helloeos -p sg，终端显示合约部署成功：

```
Reading WAST/WASM from ./helloeos/helloeos.wasm...
Using already assembled WASM...
Publishing contract...
executed transaction: 192ea1c0885e2c23c3c38e3fdb59a36f8f91447d028d1b728928ea
    ffc43da2e5  1808 bytes  818 us
#    eosio <= eosio::setcode               {"account":"sg","vmtype":0,"vmve
    rsion":0,"code":"0061736d01000000013b0c60027f7e006000017e60027e7e006...
#    eosio <= eosio::setabi                {"account":"sg","abi":"0e656f73696
     f3a3a6162692f312e3000010568656c6c6f0001047573657204 6e616d650100000...
warning: transaction executed locally, but may not be confirmed by the
    network yet    ]
```

使用 cleos 的 push action 命令来调用 helloeos 合约：

```
cleos push action sg hello '[sg]' -p sg:
```

非卖品，仅供非商业用途或交流学习使用

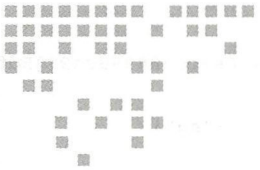
终端会显示：

```
Executed transaction: 9298e02aad5c6ac0f639d0456e9cc588968ca9735dfc4080812d5b
    5ced9 1c50c 104 bytes 992 us
#     sg <= sg::hello                    {"user":"sg"}

>> Welcome to EOS! sg
warning: transaction executed locally, but may not be confirmed by the
    network yet    ]
```

可以看，到调用合约后成功打印出“Welcome to EOS!”。

非卖品，仅供非商业用途或交流学习使用



Chapter 3 第 5 章

智能合约开发

可以说智能合约在 EOS 系统中，承载了比以太坊系统更多、更重要的任务，大部分的系统功能，诸如投票、获得资源、创建代币、转账及一些关键参数等都是通过系统智能合约实现并确定的。这样的设置为区块链系统带来了前所未有的灵活性，升级系统的诉求变得不再困难和高风险。超级节点可以通过对系统合约的提案更新，重载原来的系统合约，实现 EOS 系统升级。智能合约是 DApp 的核心，对于 DApp 开发者来说，掌握基于区块链的智能合约设计理念显得尤为重要，智能合约的设计逻辑决定了 DApp 后续是否能够安全、合理地运行并且被区块链社区所认可。

考虑到当前并没有对智能合约进行很全面的介绍，本章将首先从智能合约的定义、发展历史、设计原则、应用场景四方面出发，对智能合约进行详细介绍，旨在培养读者的智能合约开发思维；接下来会详细介绍 EOS 系统中的智能合约开发细节，包括合约文件、虚拟机、依赖库文件以及三个实战案例，通过这些内容，帮助读者快速掌握智能合约开发技巧；最后我们会介绍一些智能合约开发中常见的 bug，以及相应的 debug 思路。

5.1 智能合约简介

“究竟什么是智能合约？它们和真实世界中的传统合约是什么关系？智能合约是怎么运行的？智能合约到底能用来干什么？”，这四个问题应该是初入智能合约开发领域

非卖品，仅供非商业用途或交流学习使用

的读者最常面临的四个问题。首先将技术细节部分放在一边，本节将从智能合约的定义、发展历史、设计原则、应用场景四方面出发，详细介绍智能合约这一概念，培养读者的“去中心化逻辑”。

5.1.1　智能合约定义

在这里首先给出智能合约的标准定义：智能合约（Smart contract）是一种旨在以信息化方式传播、验证或执行合同的计算机协议。智能合约允许在没有第三方的情况下进行可信交易。这些交易可追踪且不可逆转。

这样说未免有些晦涩，其实智能合约是驻留在区块链上的一段代码，由唯一地址标识。智能合约包括一组可执行函数和状态变量。在对智能合约功能进行事务（Transaction）处理时执行这些函数，其中事务包括智能合约中的函数所需的输入参数。在执行函数时，智能合约中的状态变量根据函数中实现的逻辑而变化。

考虑一个众筹平台的例子，平台团队分享他们的项目并从支持者那里收钱，直到达到目标。如图 5-1 所示。

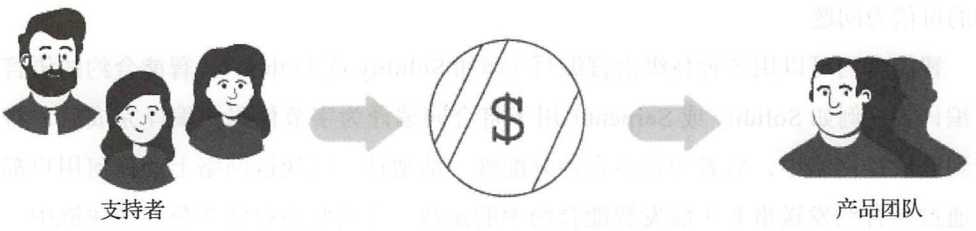


图 5-1　传统众筹平台流程

如果这样的平台是集中式的，那么它就是项目团队和捐赠者之间的第三方。这意味着双方都需要信任这个第三方，而且在现实世界中，项目方还要向这个第三方支付额外费用作为中间人担保费。智能合约的出现为解决这些问题带来了新的思路。

更确切地说，我们可以编制一份智能合约来接收资金，直到达到目标。如果项目在截止日期之前获得全额资助，则自动筹集的资金将转入产品团队。如果项目失败，资金将自动返回给支持者，如图 5-2 所示。

由于智能合约存储在区块链内，其中所有数据以分布式方式存储，因此没有人可以控制钱。在分散的商业模式中，智能合约可以取代任何其他可信赖的第三方。

非卖品，仅供非商业用途或交流学习使用

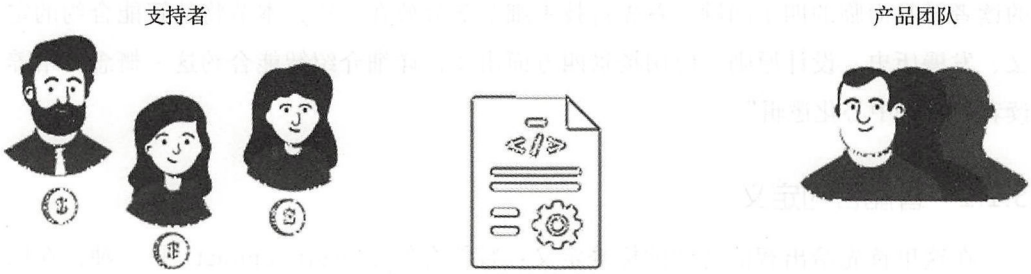


图 5-2 基于智能合约的众筹平台流程

那么，为什么会产生智能合约这样的设计呢？我们知道比特币提供了一个既不富有表现力也不方便用户的基本脚本系统。学术界和工业界的一系列工作都试图以改进比特币的脚本语言的方式设计各种智能合约应用程序。由于比特币脚本语言的表现力的基本限制，改进语言不仅耗费时间，而且还可能导致脚本执行时占用了过多的链上计算资源。相比之下，如果它们是在通用图灵完备的编程语言（以太坊是第一个创建这种语言 Solidity）之上构建的，那么许多相同的任务将更容易编程且更高效；另外人们可以将智能合约视为一个特殊的自执行的“受信任的第三方”，从而避免中心化系统中的可信力问题。

智能合约可以用各种高级语言编写（例如 Solidity 或 Python）。智能合约的语言特定编译器（例如 Solidity 或 Serpent）用于将合同编译为字节代码。编译完成后，将上传到区块链网络中，后者为智能合约分配唯一的地址。区块链网络上的任何用户都可以通过向合约发送事务来触发智能合约中的函数，合约状态存储在公共区块链中。智能合约的代码程序由区块链中的维护节点（矿工）网络执行，他们就执行结果达成共识，并相应地更新区块链上的合同状态。用户可以将资金或数据发送给智能合约，或从智能合约中收取资金或数据。智能合约示意图如图 5-3 所示。

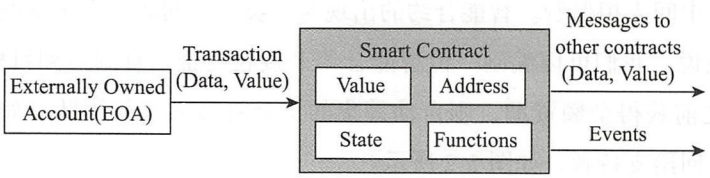


图 5-3 智能合约示意图

总结下来，智能合约具有如下三个特点：

（1）不可更改性。这一特征主要是由区块链保证的，智能合约的程序代码作为一组数据存储在区块链中，所以在部署上链后就变成固定的、不能更改的状态，所以对于智能合约来说，更新与修改操作就变得异常困难，但是在 EOS 系统中对这个问题做了优化，我们在下面的内容会有详细介绍；

（2）自执行性：智能合约的运行是交给区块链网络中的维护节点（矿工群）来执行的，对于智能合约开发者来说，可以将区块链系统当作一个执行智能合约的黑盒。部署上链后，合约将会一直执行下去，不可以中断这个执行过程；

（3）公开性：智能合约中的数据对所有用户都是公开并可读取的，这样说或许有些绝对，但是考虑到智能合约通常选择部署在公有链中，而在公有链中，链上数据对区块链网络中的每一个节点都是公开透明的，所以我们也将这一特性罗列在三大特征中。

那么智能合约是如何工作的？

在区块链系统中，智能合约可以被当作存储在区块链中的脚本（它们可以被认为与关系数据库管理系统中的存储过程大致相似）。由于它们位于链上，因此智能合约具有单独的地址。用户可以通过向指定合约发起事务从而触发对应合约中相应的函数，函数的执行将根据事务中包含的数据更新智能合约的状态，整个过程独立且自动地在区块链系统上执行（这意味着区块链中的每个节点都在自己的虚拟机（VM）中运行智能合约），整个区块链系统可以被视作一个分布式的虚拟机（Virtual Machine）。

智能合约允许我们在链上进行简单的通用计算。然而，智能合约最擅长的地方，在于管理网络上实体之间的数据交互。让我们用一个例子解释一下这句话。考虑一下 Alice、Bob 和 Carol 参与的区块链网络，以及交易 X 和 Y 类型的数字资产。Bob 在网络上部署了一个智能合约，该合约定义：

- “存款”功能，允许 Bob 将 X 资产存入合约；
- “交易”功能，允许其他用户将 Y 资产存入合约，来兑换 Bob 存入的 X 资产，兑换比例为 1：5；
- “撤销”功能，允许 Bob 撤回合同所持有的所有资产。

请注意，“存款”和“撤销”功能只有 Bob（通过他的密钥）可以调用它们（因为这必须由 Bob 来决定），我们假设这个智能合约已经部署在了某个区块链网络中，该网络上的任何用户都可以成功调用它。

仅供非商业用途或交流学习使用

接下来考虑这样一个过程：

Bob 向该智能合约的地址发送交易，调用其“存款”功能并将 3 个 X 资产转移至合约中。此交易记录在区块链中。然后拥有 12 个 Y 代币的 Alice 发送一个交易触发该合约的交易功能，交易内容为将 10 个 Y 资产兑换为 2 个 X 资产，此交易也将记录在区块链上。最后 Bob 通过交易，将带有自己私钥的数字签名发送给合约，触发“撤销”功能。合约检查签名以确保撤回是由合约所有者发起的，若通过，则将其中所有存款（1 个 X 单位和 10 个单位 Y）转回给 Bob。

总结一下智能合约的特征，有如下 8 点。

❑ 智能合约有自己的状态，并且可以对区块链上的资产进行保管。

由于智能合约在区块链上有自己的账户，而区块链支持基于账户的模型，所以从角色上来看就相当于一个自动代理人（如果我们回到共享数据库模型，合同是一个单独的“用户”实体，可以拥有、删除和创建）。在上面的示例中，它可以保存资产 X 和 Y。

❑ 智能合约允许我们用代码表达业务逻辑。

例如上面示例中的“交易功能”，每收到 5 个 Y 资产将交换 1 个 X 资产。

❑ 正确编写的智能合约应描述合同的所有可能结果。

例如，可以修改上述“交易”功能，以便拒绝兑换数量不是 5 的倍数个 Y 资产的交易，如当 Alice 将 12 个 Y 资产全部触发合约中的交易函数时，函数予以拒绝，或者可以在编写交易函数时添加“找零功能”，当 Alice 试图兑换 12 个 Y 资产时，智能合约将返回 2 个 X 资产和 2 个 Y 资产。

❑ 智能合约中交易的对手双方建立的关系是由数据驱动的。

这句话的意思是，合约的部署者不需要实现确定合约的对手方是谁，只需要确定相关的权限确认逻辑，保证后续对手方按照预先设定好的规则行事即可。在上面的示例中，Bob 部署了一份智能合约，其实相当于说：“如果你向这份合约发送合理的数据（5 倍数的 Y 资产），你将自动成为合约的对手方，并自动执行合约内容（返回对应数量的 X 资产）；

❑ 智能合约由发送到其地址的消息 / 交易触发。

我们可以将智能合约想象成一个状态机，它的状态变化需要通过满足预先条件的动作来触发，当状态机接收到这样的动作后，会触发执行并完成状态更新，这样的动

仅供非商业用途或交流学习使用

作在区块链中通过交易的形式实现，而智能合约的地址相当于状态机的 ID，起标识作用。

- 智能合约是确定性的，相同的输入将始终产生相同的输出。

如果一个人写了一个非确定性合同，当它被触发时，它将在网络上的每个节点上执行，并可能返回不同的随机结果，从而阻止网络就其执行结果达成共识。在一个正确构建的区块链平台中，编写非确定性智能合约是不可能的。

- 智能合约保存并运行在区块链上，因此每个网络参与者都可以检查其代码。
- 由于与合同的所有交互都是通过区块链上的签名消息进行的，因此所有网络参与者都可以获得合同操作的加密可验证跟踪。

支持比特币式交易（UTXO）的区块链可以在不相互信任的交易对方之间进行资产转移。然而，支持智能合约的区块链在技术上就显得更进一步，它允许在相互不信任的交易对方之间进行多步骤流程。这就要求交易实体在决定参与合约之前：

- 检查代码并确定其结果。
- 由于代码已经部署在它们都没有完全控制的网络上，因此可以进行确定执行。
- 由于所有交互都经过数字签名，因此合约执行的整个流程具有可验证性，合约执行所产生的争议的可能性被消除（当智能合约的设计考虑到所有可能的结果时），因为参与者不能对他们参与的这个可验证过程的最终结果持不同意见。

智能合约作为自主行为者运作，其行为完全可以预测。因此，区块链系统允许智能合约通过合理的逻辑函数修改属于合约本身的链上数据，即具备了这部分数据的控制权。通过智能合同也产生了“分布式自治组织”（DAO）的概念，都是指通过智能合约确定一系列公开公正的规则，可以在无人干预和管理的情况下自主运行的组织形式。这些规则往往会以开源软件的形式出现，每个人都可以通过购买该组织的股份权益，或者提供服务的形式来成为该组织的参与者。从某种角度来看，DAO 就像一个全自动的机器人，当它全部的程序设定完成后，它就会按照既定的规则开始运作。值得一提的是，在运作的过程中，它还可以根据实际情况不断地自我维护和升级，通过不断地自我完善来适应它周围的环境。

事实上，DAO 的形态非常广泛，它可能是某种数字货币，也可能是一个系统或者机构，甚至可能是无人驾驶汽车。它们为客户提供有价值的服务。这种服务可以是货币传输（如 Bitcoin）、应用平台（Ethereum）、域名管理系统（Namecoin），或者是任意

仅供非商业用途或交流学习使用

一种其他的商业模式，这些商业模式显然更像是特定机构的股票而不是单一的货币。每个 DAO 都有其自己的条款和条件。你将永远有权查看你拥有的、可支配的、数字货币形式的 DAO 股份，并且有可能从中获得股息。

5.1.2 智能合约历史

和人工智能一样，其实智能合约这一概念很早就已经被提出了，Nick Szabo 在 1994 年引入了这个概念，并将智能合约定义为“执行合同条款的计算机化交易协议”。Szabo 建议将合同条款（抵押品、债券等）转换为代码，并将它们嵌入到可以自运行的资产（硬件或软件）中，以便最大限度地减少交易方之间对可信中介的需求，以及第三方恶意或非故意的执行错误。区块链技术的出现为智能合约带来了真正的润土，使设计具有可用性的智能合约变成了可能。下面我们将介绍智能合约的发展历史，并希望读者在这个过程中能够真正了解智能合约的使命与设计逻辑。

1. 最初的智能合约概念

随着以太坊以及 Solidity 和 Serpent 等编程语言的发布，智能合约技术蓬勃发展，这使得智能合约的构建和部署变得更加简单。不过，这个想法已经存在了很长时间。以数字货币研究著称的密码学家 Nick Szabo 早在 1995 年就写了一篇关于智能合约的文章，并在《 Extropy 》杂志上发表，文章中对智能合约的优势、实现途径与具体要求做了详细并具有先见性的设计与说明，可以说其中的设计思想对当前基于区块链的智能合约设计依然具有指导性作用。

Szabo 将合同定义为“在思想会议中同意的一系列承诺这是传统的正式化关系的方式。”此类合同是自由市场经济的支柱，可用于商业关系、婚姻和政治。在他 1995 年的文章中，Szabo 预测数字革命将彻底改变人类签订合同的方式，他甚至质疑我们的传统合同是否会继续在网络时代使用。Szabo 早就看到计算机凭借强大的计算能力，会极大地降低运行算法的成本，并且相信算法最终将被开发用于他所谓的“智能合约”。他将此定义为“以数字形式指定一组承诺，包括各方在其他承诺上执行的协议”。

智能合约将改善五个基本合同目标的执行，Szabo 将其描述为可观察性、可验证性、相互关系性、可执行性和隐私性。在以下各节讨论的用例中，根据 Szabo 的智能合约将使双方能够观察对方履行合同状态，验证合同是否以及何时履行，保证只有完

仅供非商业用途或交流学习使用

成合同所需的细节才能透露给双方并自我执行，以消除维持合同的时间。接下来，我们详细介绍一下 Szabo 对智能合约的定义与设计思想，了解这些知识对于设计优质智能合约以及优秀公链系统有着何等至关重要的意义。

1）什么是智能合约

我们可以将合约对标到真实世界中的合同来理解，合约是一种在进行了一定的“思想交流”后，商讨人之间达成一致的承诺，是将关系正式化的传统方式。虽然合同主要用于商业关系，但它们也可能涉及诸如“婚姻”之类的个人关系。合约在政治中也很重要，不仅因为“社会契约”理论，而且因为合约执行传统上被认为是政府的基本职能。

无论是由政府强制还是其他方式，合约都是自由市场经济的基本组成部分。经过许多世纪的文化演变，出现了契约的概念和与之相关的原则，并被编入法律条文中。算法信息理论表明，这种演化结构的重新计算成本往往过高。如果我们从头开始，利用理性和经验，可能需要几个世纪来重新开发出像现代自由市场一样的产权等复杂想法。

合同法的成功，加上替换合同的高成本，使得在适当情况下保留和利用这些原则是值得的。然而，数字革命正在彻底改变我们可以拥有的各种关系。我们来之不易的法律传统的哪些部分在网络时代仍然有价值？将这些普通法原则应用于我们的在线关系设计的最佳方法是什么？

计算机大大降低了算法的运行成本，并且可以更快地传输更大和更多的复杂消息。此外，计算机科学家和密码学家最近发现了许多新的且非常有趣的算法。结合这些消息和算法可以实现各种新协议。

现在，数字革命使新机构和构成这些机构的关系正式化的新方法成为可能。这些表示方法就是一种“智能合约”，因为它们比无生命的纸质合约更具功能性，且不局限于使用人工智能。简单说，智能合约是一组承诺，以数字形式指定，包括各方履行这些承诺的协议。

2）一种嵌入世界的智能合约

智能合约的基本思想是，许多合同条款（如留置权、保税、产权划分等）都可以嵌入我们处理的硬件和软件中，以致违约对于破坏者来说价格昂贵。举一个典型的现实生活中的例子，我们可以认为是智能合约的最初实现，那就是一个可能不太起眼的自动售货机。在有限数量的潜在损失中（投币孔中的金额应小于破坏机器的成本），机器接收硬币，并通过简单的机制，分配变化和产品公平（这属于一个有限状态机设计

仅供非商业用途或交流学习使用

中的初级问题）。然而智能合约超越了自动售货机，并可以嵌入到各种有价值且受数字手段控制的财产中。智能合约以动态、主动执行的形式引用该财产，并在主动措施不足的情况下提供更好的观察和验证。自动售货机（如电子邮件）在自动售货公司和客户之间实施异步协议的地方，一些智能合约需要两方或多方之间的多个同步步骤。

其他智能合约的先行者包括 PoS（销售点）终端和卡、EDI（电子数据交换，用于订购和大公司之间的其他交易）、SWIFT、ACH 以及 FedWire 网络，用于银行间转账和清算付款。这些实施商业安全模式，往往很少注意到各方的合同需求和义务。

3）智能合约的鲁棒性

一个优秀的智能合约设计者在设计过程中，应该仔细考虑可能出现的如下三方面攻击的抵抗模型，以提升智能合约的鲁棒性。

- ❑ “天真”违约：这种违约行为的出现通常由于智能合约逻辑不够严谨，对可能造成的违约行为定义不明确或惩罚力度不足，从而导致缔约双方由于对违规行为的后果缺乏充分的理解（天真行为）而产生违约行为。
- ❑ 激励相容（incentive compatible）缺失导致的理性违约：这种违约行为指的是由于智能合约中对于利益分配模式设计时没有充分考虑激励相容而导致的缔约双方出于对自身利益最大化的角度考虑而产生的违约行为。首先来解释一下“激励相容”的概念。在市场经济中，每个理性经济人都会有自利的一面，其个人行为会按自利的规则行为行动；如果能有一种制度安排，使行为人追求个人利益的行为，正好与企业实现集体价值最大化的目标相吻合，那这一制度安排就是“激励相容”。现代经济学理论与实践表明，贯彻“激励相容”原则，能够有效地解决个人利益与集体利益之间的矛盾冲突，使行为人的行为方式、结果符合集体价值最大化的目标；对标到智能合约领域，即缔约双方自身利益与整体利益的两个目标函数一致化，拥有信息优势的一方按照合约另一方的意愿行动，从而使双方都能趋向于效用最大化。说白了，就是没有人可以通过损害集体利益去实现自己利益的最大化。
- ❑ 恶意的、复杂的攻击行为：这种违约行为一般比较少见，通常是由于恶意第三方攻击者可以并且愿意牺牲大量资源攻击合约，如“51% 攻击”等。

4）合约设计的几个基本原则

考虑传统真实世界中的合约，其有效性与安全性由以下三点保证：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 合约是由某司法系统或强有力的第三方保证的，司法系统决定执法机构应采取哪些实际步骤（包括逮捕、没收财产等）以应对违反合约的行为，这通常也被称作“反应式安全机制”。
- 通过对执行动作的真实记录使合约执行结果可以被验证，例如通过在摄像机上记录违规行为，或在合约文件上签名，法庭判断违约索赔诉求时有据可依。
- 正在进行的合同是可以被观察的，以便发现违约的第一个迹象时及时处理，从而将损失降至最低，这也是一种反应性的安全形式。

总结下来一个有效且安全的真实合约需通过一定的手段，使违规的成本高于收益，通过对真实合约的分析，可以提炼出契约设计的五个基本目标。

- 可观察性：即缔约人观察彼此履行合同的能力，或向第三方仲裁机构证明其表现的能力。
- 客观可验证性：缔约人向仲裁机构证明合同已被执行或违反的能力，或仲裁员通过其他方式找到这一点的能力。可观察性和可验证性还可以包括区分故意违反合同和“善意”错误的能力。
- 合约相对性：合约关系理应不涉及第三人，这指的是合约的效力范围仅限于合约当事人之间，合约当事人一方只能向合约的另一方当事人基于合约提出请求，而不能向与其无合约关系的第三人提出合约上的请求。同时，因合约而产生的违约责任只能在合约当事人之间发生，合同关系以外的第三人不能基于该合同而承担违约责任。
- 目标的可执行性：同时最大限度减少执行需求。智能合约需要通过实现内置用户声誉、激励模型、合约“自执行”协议和可验证性保证其可执行性。换句话说，这应该是智能合约的最终要求。
- 隐私性：智能合约通常涉及受信任的第三方，例如参与履约的中间人以及仲裁员，该仲裁员被用来解决因违约而引起的纠纷。隐私性意味着智能合约的验证应该最大限度地减少对第三方的依赖程度，尽可能少的暴露合约执行的细节数据，这在数据时代中具有深远的意义。由于可验证性和观察性通常要求智能合约必须信任第三方仲裁员，并允许他们知晓合同执行过程中发生的大部分动作，这样的模式对于合约双方的隐私保护显然是不合适的，因此保证智能合约的隐私性在当前时代就显得尤为重要。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

5）尼克萨博预测的智能合约的应用领域

尼克萨博在提出智能合约概念时，对其可能的应用领域做了预测，并且我们可以看到，通过区块链实现的智能合约正在一步步实现这些预测。

跨国小企业

在 1995 年的市场中，跨国企业而非当地企业进行了大部分国际贸易。Szabo 预测，在未来的市场中，将出现一种小型跨国企业，其规模与 1995 年的本地企业相似。他认为这种变化可能会像电信业一样迅速发生。

对于 1995 年的本地企业来说，随着光纤电缆等创新技术的出现，通信和运输成本已经变得非常低。尽管如此，法律问题仍然为想要进入全球竞争领域的本地企业创造了障碍。Szabo 在他的文章中预测，智能合约将克服这一障碍，大幅降低法律成本以及遵守当地法律法规的要求。

虚拟角色

Szabo 还建议人们可以创造不同的虚拟角色，以迎合一个人的个性或生活方式的不同部分。人们可能在工作、家庭、朋友关系等中扮演不同的角色。每个角色都可以是独一无二的，并且只共享该人在他或她生活的这一部分中感到舒适的数据。

Szabo 将“ nym”定义为“仅将少量关于某人的相关信息链接起来的标识符，通常是 nym 持有者认为与特定组织或社区相关的信息”。nyms 可以包含昵称、品牌名称或别名。

Szabo 将一个真正的名称定义为“一个标识符，用于链接关于某个人的许多不同类型的信息，例如他们的出生名称或社会安全号码”。Szabo 预测，知道某人的真实姓名可以为一个人提供巨大的经济价值；他引用的一个例子是允许组织发送目标产品信息，因为知道该人感兴趣。

虚拟世界中每个人的目标都是创建一个“声誉良好的名称”，这个名称将是一个带有正面属性且受到高度重视的 nym 或真名。Szabo 预测，有一天人们会出售虚拟角色，类似于公司销售品牌的方式。

智能财产

随着智能合约使用的增加，Szabo 相信我们会开始将智能合约嵌入我们的物业中，他称之为“智能财产”。这些嵌入式智能合约会自动授予物业或客人的合法拥有者权限，取决于合同的参数。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Szabo 提供了一个租赁汽车的例子，如果承租人没有及时付款，其嵌入式合同可以自动将汽车的控制权交还给贷款人。通过这种方式，智能财产只会为真正的所有者提供价值，并可以消除被盗的可能性。

合约可视化

通过智能合约，有必要与所有相关方沟通协议的确切条款。精细打印或错误表示的信息是传统合同的一个巨大问题，而智能合约可以在代码中写入“精细打印”，这些代码在交易中被若干方之一隐藏。例如，杂货店的 PoS 系统，智能合约的基本版本，可以收集关于客户的数据，例如他们的名字，并且在客户不知情的情况下将其存储在数据库中。

Szabo 的主流可用性解决方案是智能合约应该集成交易元素的可视化表示。这将允许各方不仅仅依赖于查看代码来同意合同的语义，并且可以消除代码中存在隐藏操作的任何风险。

2. 区块链是智能合约实现的沃土

上一节我们将 Nike Szabo 所提出的智能合约的概念进行了详细介绍，那么为什么直到以太坊的出现，这一概念才会被人们广泛接受呢？我们重新审视一下 Szabo 所设计的智能合约，可以发现这个设想想要实现，前提有两点：

- 需要一个值得信任的第三方主体来运行这个智能合约，执行过程必须严格遵循合约逻辑，而且整个执行过程必须是可追溯并且完全公开透明的，若仅仅是让合约参与方当中的某一方来单独运行势必会产生不公平，那么我们考虑真实世界场景，两个公司想要签署一个商业智能合约，就必须找到一个共同信任的主体作为智能合约的执行者，这个主体需要且必须知道该合约中的任何细节并且执行合约中的每一个动作，这就产生了严重的信任危机问题。很难找到这样一个具有强有力的第三方主体运行智能合约，或者说找到这样一个主体所需的成本很高，智能合约所带来的优势并不足以弥补这些成本。
- 智能合约所涉及的资产必须是数字化的，在当时的时代中，现实生活中的资产数字化的难度较高。

所以说这个优秀的想法一直被搁置，直到区块链的出现！我们知道在区块链出现之前，任何数字资产都需要有一个中心化的第三方做信任背书，这个第三方可以是国家或者大型机构，用户可以通过这个可被信任的第三方完成数字资产与真实资产之间的兑换，这就产生了上述问题。然而，比特币系统第一次实现了一种不需要第三方背

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

书的去中心架构化的数字资产，这使得智能合约的实现成为可能，而且比特币系统中的输入与输出脚本已经具备了智能合约的影子，这些都给以太坊的开发提供了一定的思路，以太坊是第一个支持智能合约的公有区块链系统，他的设计者 VL 旨在通过内置虚拟机沙盒（EVM）的设计，将区块链系统从原来的价值交换网络变成一个去中心化的服务器网络，所以说以太坊的出现标志着区块链正式进入 2.0 时代，用户可以将自己开发的智能合约部署在区块链上作为某项目中用户数据状态机，并搭配一些中心化模块对外提供服务，这也是最简单的去中心化应用（DApp）架构。这样的设计使得开发者可以通过区块链对外提供去中心化服务，极大地扩展了区块链的功能。

5.1.3 智能合约优劣势

区块链中智能合约的优势：

- 没有中介机构。如前所述，缺乏中介使得流程更容易、更快捷。现在没有必要寻求律师的帮助，一切都遵循预先编写的代码。
- 安全和保密。所有合同都以加密形式存储在区块链中。该系统的目标是没有人可以改变智能合约或更换数据。此外，私钥用于签名，提高了智能合约运行的安全级别。
- 执行速度快。纸上工作需要很长时间。区块链智能合约的使用大大加快了结束交易的过程。
- 成本较低。智能合约将中间商排除在业务流程之外。一旦编写，该程序可用于不同用户的不同交易。

任何事物都具有两面性，不可避免地，基于区块链的智能合约也存在缺点：

- 编码的难度。在现实生活中你可以从律师那里得到一份具有法律效益的合同样板。有很多这样的服务，你只需联系法律机构即可获取到必要的文件。在区块链智能合约的情况下，律师的角色由开发人员执行，所以这要求开发人员具有一定的法律知识。目前，此类专家的数量有限，并非所有专家都具备相关的技能水平。
- 高风险性。智能合约中的错误可能导致巨大的资产损失。由于智能合约往往涉及价值的分配关系等经济因素，任何开发人员在编写智能合约过程中犯的错误，都可能导致资金损失或允许黑客窃取资金。

非卖品，仅供非商业用途或交流学习使用

- 灵活性。智能合约没有灵活性。区块链的技术以不允许更改数据的方式构建。以前创建的区块链智能合约不能替换任何数据或进行编辑。
- 法律规定。到目前为止，还没有法律可以根据区块链智能合约来规范这种关系。由于政府尚未通过必要的法律，法院将无法接受相关申请。

5.1.4　智能合约应用场景

本节主要介绍 10 种智能合约的典型应用场景，这 10 种应用场景分别隶属 10 个不同的领域。

1. 数字身份智能合约

智能合约可以使个人直接拥有和控制包含声誉、数据和数字资产的数字身份，这在传统互联网领域中是很难实现的。采用智能合约管理用户的数字身份允许用户可以选择向交易对方披露哪些个人数据，且这个过程需要支付一定费用；对于企业来说，也可以无缝地了解他们的客户。相关架构如图 5-4 所示。

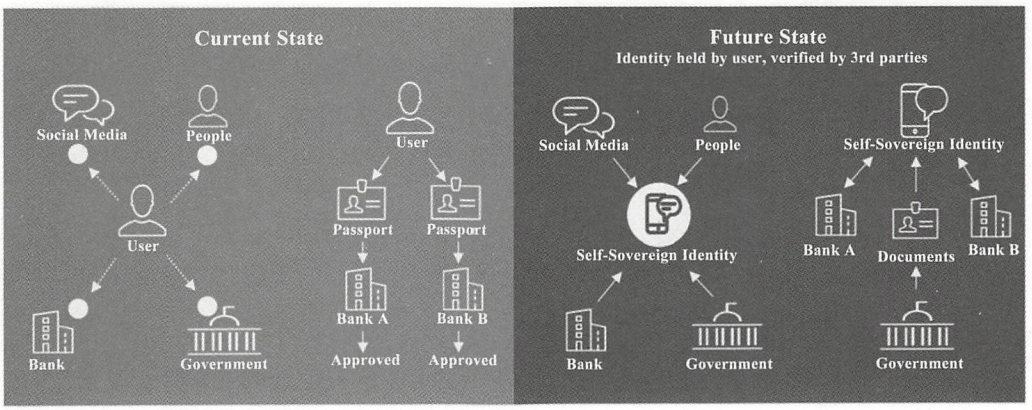


图 5-4　数字身份智能合约示意图

当前系统的挑战与局限：

- 对于企业来说，了解你的客户（KYC）需要复杂且昂贵的流程，而且这个流程缺乏完整性。
- 由于个人对可信赖的第三方的依赖，对潜在数据泄漏的控制有限。
- 保存用户数据的服务器通常都成为黑客发起单点攻击的目标。

非卖品，仅供非商业用途或交流学习使用

使用区块链的优势：

- 用户个人可以直接拥有和控制个人数据（例如能够安全地向各个交易对方披露个人数据）。
- 数据交易对方不需要持有用户个人信息等敏感数据来验证交易、减少责任，同时促进无摩擦的 KYC，身份的验证交由智能合约进行。
- 提高流程的合规性、弹性和互操作性。

应用过程中的注意事项：

- 需要促进在法律框架内接受数字提供的证明，并建立对智能合同安全性的信任。
- 需要制定协议和标准以提供相关各方的互操作性。

2. 证券型智能合约

通过部署智能合约，可以简化证券产品管理，并在证券管理链中规避中间人作恶风险。智能合约可以实现股息，股票分割和负债管理的自动支付，同时降低交易对手的操作风险。图 5-5 给出了相关架构。

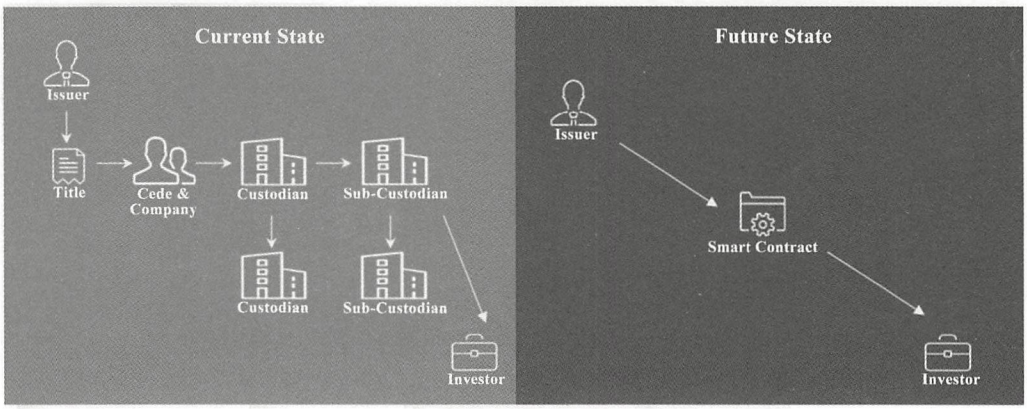


图 5-5 证券型智能合约示意图

当前系统的挑战与局限：

- 对于待发行证券的公司在发行证券前需要提供良好信誉证明，但在中心化方案中很难做到。
- 中介机构增加了成本、交易对手风险和延迟。

使用区块链的优势：

非卖品，仅供非商业用途或交流学习使用

- 通过智能合约管理证券发行可以带来数字化端到端工作流程的好处。
- 证券的即时结算性，结算周期为 T+0 模式。
- 便于自动支付股息和股票分割，同时实现更准确的代理买卖。
- 消除中介机构带来的交易操作风险。

应用过程中的注意事项：

- 需要政府部门承认这种模式的合法性。
- 虽然发行人会很乐意了解谁拥有了他们的证券，但一些买方公司（例如激进投资者）会谨慎地保护这些信息，所以需要考虑这部分用户的需求。

3. 贸易融资智能合约

智能合约可以通过更快的信用证明和贸易支付启动模式来促进并简化国际货物转移流程，同时实现更高的金融资产流动性。这种模式为买家、供应商和金融机构提供风险缓解并且改善融资和流程效率。示意图如图 5-6 所示。

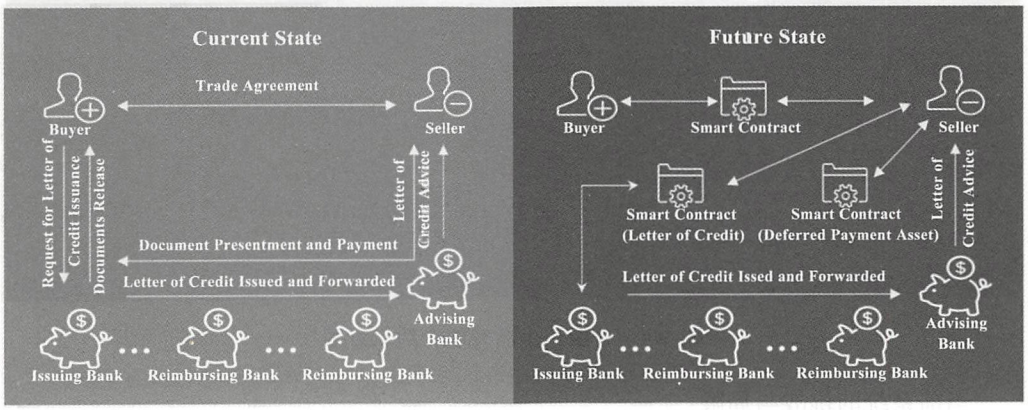


图 5-6　贸易融资智能合约示意图

当前系统的挑战与局限：

- 繁重的协调工作，带来了耗时且昂贵的信用证明发放流程。
- 真实世界中的物理凭证管理可能造成延迟发货收货的问题。

使用区块链的优势：

- 通过自动化信用监控流程，更快地批准和支付启动。
- 提高创建、修改和验证贸易、产权、运输相关合同协议的效率。

非卖品，仅供非商业用途或交流学习使用

❑ 易于转移和减少欺诈行为，增加了金融资产的流动性。

应用过程中的注意事项：

❑ 必须实施全行业的智能合约模板和程序标准，以便被更广泛地接受和采用。

❑ 必须确定潜在智能合约执行失败的法律影响（特别是对于违约和争议解决）。

❑ 必须实现结算系统、离线生态系统和技术先决条件（例如物联网）的完全整合。

4. 金融衍生品的智能合约

智能合约可以简化交易后的流程，消除每个交易对手履行的验证交易，在适当的贸易活动中减少重复过程。它们制定了一套标准的合同条款，并优化场外交易衍生产品的交易后处理过程。它们还能够实时评估位置，以便监控和减少错误。金融衍生品智能合约示意图如图 5-7 所示。

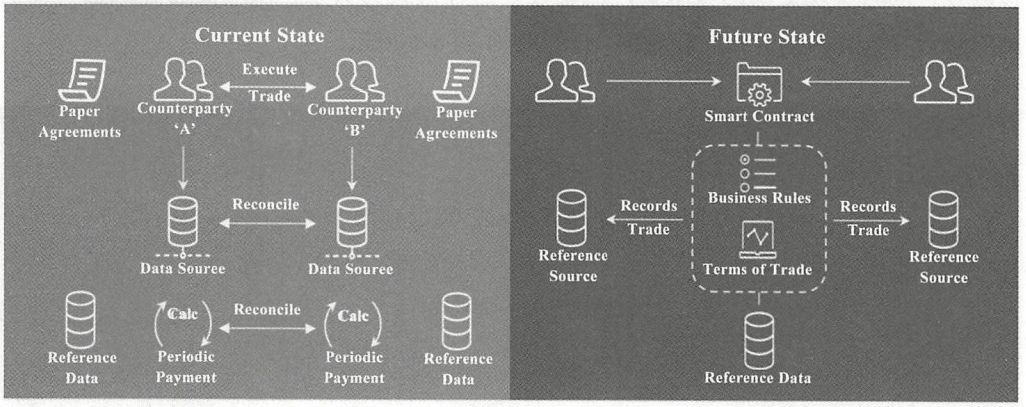


图 5-7 金融衍生品智能合约示意图

当前系统的挑战与局限：

❑ 由于大多数场外交易衍生品通常被各个发起方独立管理，因此造成了冗余且耗时的流程。

❑ 基于纸质的交易协议，包含条款、贸易协议和交易后确认流程严重低效。

使用区块链的优势：

❑ 在执行触发的贸易事件处理（例如定期付款）时自动解决债务。

❑ 可以自动化外部事件处理（例如信用），提高效率。

❑ 实现了对金融衍生品的全程评估，同时减少错误与纠纷。

非卖品，仅供非商业用途或交流学习使用

应用过程中的注意事项：

- 建立区块链网络及其智能合约的合规化治理模式。
- 需要充分考虑场外衍生品生命周期中所需外部数据源的真实性。

5. 财务数据记录智能合约

金融机构可以利用智能合约来准确、透明地记录财务数据。智能合约可以跨组织实现统一的财务数据，改善财务报告并降低审计和保证成本。智能合约可以为进入金融交易的实体准确记录财务数据。财务数据记录智能合约示意图如图 5-8 所示。

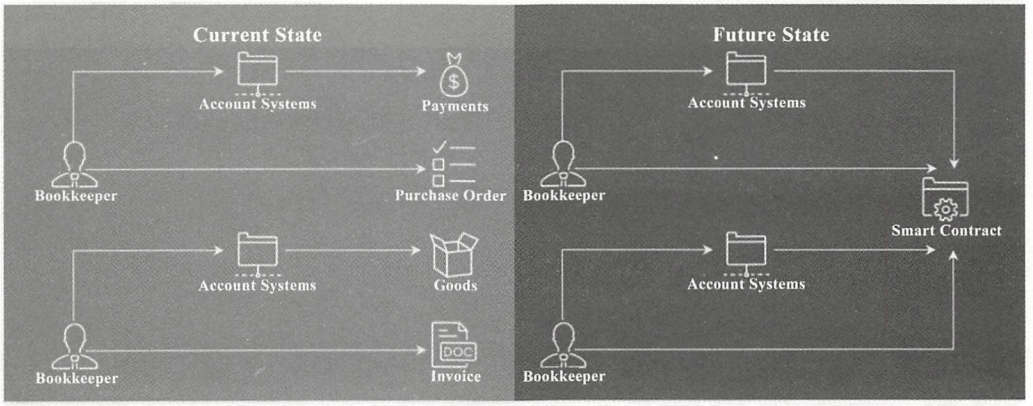


图 5-8　财务数据记录智能合约示意图

当前系统的挑战与局限：

- 会计系统容易出现欺诈和错误，因为它们由某人直接控制。
- 由于每家公司都有自己的基础设施，因此造成管理效率低下。
- 处理来自不互操作的系统的交易所需的重要人力资本 / 中间件。

使用区块链的优势：

- 提高交易数据的完整性和透明度，提高市场稳定性。
- 通过跨多个组织的成本分摊减少会计信息系统的支出。
- 由于财务数据透明性与共享性的增加，提高了对各方资本的洞察力。

应用过程中的注意事项：

- 需要开发一个门户网站，简化智能合约，促进和报告金融交易。
- 需要为标记化资产设计一套标准。

- ❑ 需要解决区块链网络与传统系统之间的互操作性。
- ❑ 需要设置证明机构以审计金融智能合约。

6. 抵押智能合约

智能合约可以通过自动连接各方来完成抵押合同的执行，从而提供流畅且不易出错的体验。智能合约可以在贷款支付时自动处理付款并从土地记录中释放留置权。它们还可以提高所有参与方的记录可见性，并有效促进付款的跟踪和验证。它们减少了人工操作过程中的错误和成本。为了实现这一点，数字身份的核验非常关键。抵押智能合约示意图如图 5-9 所示。

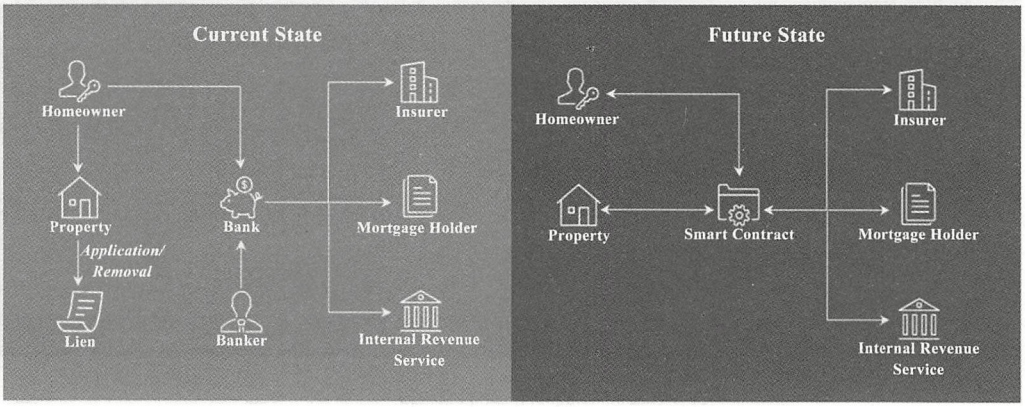


图 5-9 抵押智能合约示意图

当前系统的挑战与局限：

- ❑ 流程过于复杂，包括支付申请、更新余额、等冗余流程。
- ❑ 由于对方需要了解抵押者的真实身份而引起的隐私问题。

使用区块链的优势：

- ❑ 可以在偿还抵押贷款时自动从土地记录中释放留置权。
- ❑ 提高服务商记录对所有相关方的可见性，实现支付验证和跟踪。
- ❑ 通过消除手动过程降低成本和错误。

应用过程中的注意事项：

- ❑ 需要开发智能合约，用来作为借款人支付账户，实现支付账户和房地产业务记录服务之间的交互接口。

- 必须成功实现基于智能合约的数字身份才能实现这种模式。
- 抵押人与有关各方之间采用公钥基础设施验证彼此身份。

7. 土地权记录智能合约

通过智能合约促进财产转移，可以减少欺诈倾向，同时增加对身份的信心。这些交易可以通过提高效率、完整性和透明度来实现，从而降低成本并增强流动性。通过智能合约实现的财产转移可以阻止欺诈并提高交易的完整性、效率和透明度。土地权记录智能合约示意图如图 5-10 所示。

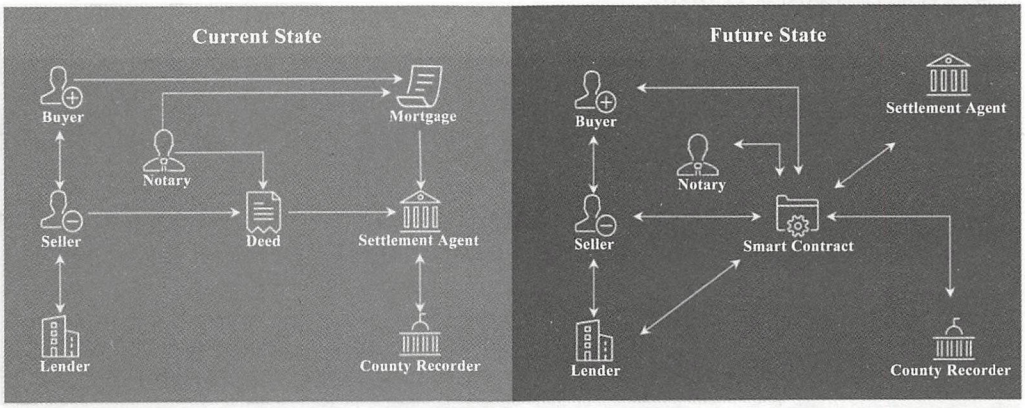


图 5-10　土地权记录智能合约示意图

当前系统的挑战与局限：

- 由于基础设施不兼容导致资本密集。
- 纸质身份验证和签名过程效率低下。
- 手动流程会延迟关闭、托管和记录流程，并可能导致纸质文档更改或丢失。
- 由于记录系统不统一，可能存在一地多用的情况。

使用区块链的优势：

- 可以使流程简化，降低审计成本。
- 可以保证土地交易记录的完整性。
- 减少土地所有权欺诈。
- 增强流动性。

应用过程中的注意事项：

- ❑ 参与实体必须使用标准化记录格式（如数据元素和电子签名字段）。
- ❑ 必须开发通用协议，以便与各方和电子记录文件格式进行通信。
- ❑ 必须接受联合身份凭证。

8. 供应链管理智能合约

智能合约可以提供供应链中各个环节的可见性。随着产品从工厂车间移动到商店货架，物联网设备可以写入智能合约，提供企业整个供应链的实时可见性。通过智能合约实现的供应链可视性扩展，可以跨品牌、零售商、物流跟踪货物流通情况。供应链管理智能合约如图 5-11 所示。

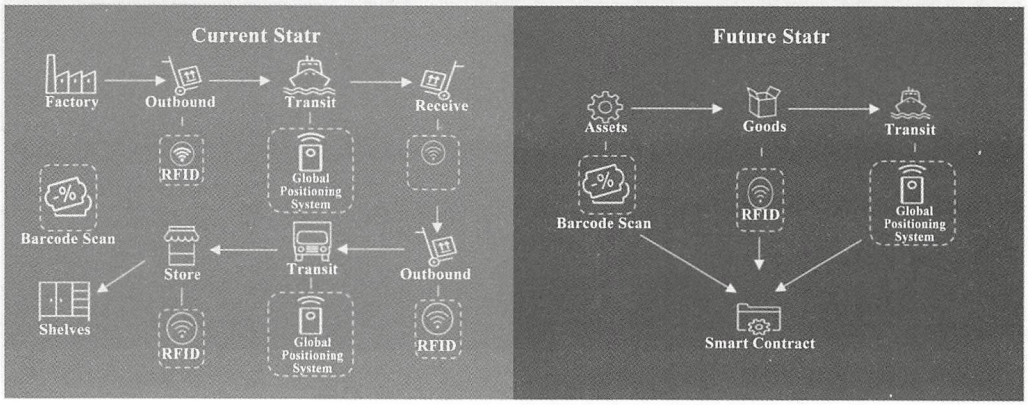


图 5-11 供应链管理智能合约示意图

当前系统的挑战与局限：

- ❑ 由于孤立的数据捕获仅与相关方的信息共享，造成供应链流程可见性低。
- ❑ 需要对捕获的数据进行类似格式化以提取其中关键值。

使用区块链的优势：

- ❑ 简化复杂的数据交互系统。
- ❑ 实现精细级别的库存跟踪和交付保证，可能会改善供应链融资、保险和风险。
- ❑ 增强追溯和验证的真实性，以降低欺诈和盗窃风险。

应用过程中的注意事项：

- ❑ 必须实施可信的 oracles 以提供供应链企业实体的有效注册。
- ❑ 身份必须全部进行登记和备案，包括机构、个人、传感器、设施和货物。

9. 汽车保险智能合约

目前，汽车保险索赔流程是脱节的，但通过智能合约可以显著改善流程。智能合约记录了所有司机驾驶记录和行车报告，使物联网车辆能够在事故发生后不久执行初始索赔。智能合约实现的自动保险索赔能够相互通信并评估和验证自身状况的车辆进行即时处理、验证和支付。汽车保险智能合约如图 5-12 所示。

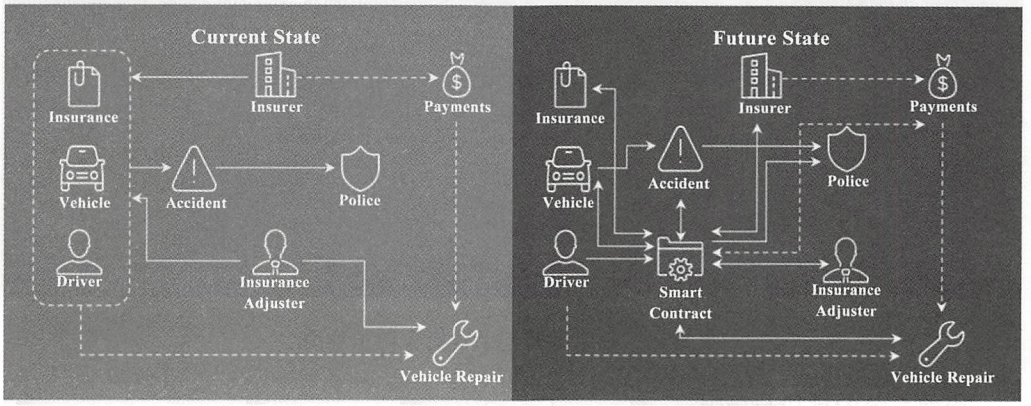


图 5-12　汽车保险智能合约示意图

当前系统的挑战与局限：

- 多种形式的报告和数据源会增加错误倾向并浪费时间和资源。
- 由于保险提供商需投入人力资源来核实重复的记录和报告，因此造成效率低下。

使用区块链的优势：

- 每个保单持有人的数据库中可以包括全球驾驶记录、车辆类型和事故报告历史。
- 使用传感器记录车辆行驶全过程动作，在损害评估可以获得准确的事故分析报告。
- 通过减少重复工作来节省更多成本。

应用过程中的注意事项：

- 需要实施分布式自治政策（DAP）提高系统的可扩展和自动执行性，进而提高效率。
- 必须实现创新的跨行业协作，解决这种系统中技术、财务和监管方面的挑战。

10. 临床试验记录智能合约

通过提高跨机构的可见性，临床试验可以从智能合同中受益。智能合约可以保证

数据的隐私性，可改善机构之间的数据共享模式，同时患者可以直接控制自身健康数据，以决定想和谁共享这些数据。通过智能合约提高可视性可以通过增加生态系统参与者的数据共享来简化临床试验过程。临床试验记录智能合约如图 5-13 所示。

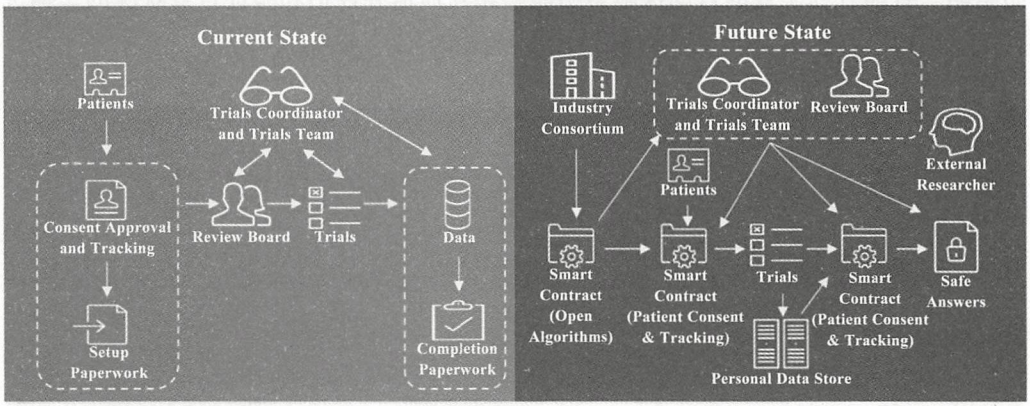


图 5-13 临床试验记录智能合约示意图

当前系统的挑战与局限：

- ❑ 由于临床试验共享数据的摩擦导致对流行病的反应延迟。
- ❑ 由于缺乏一致的同意管理，患者参与有限。
- ❑ 由于共享数据集而导致患者隐私和重新识别。

使用区块链的优势：

- ❑ 通过简化试验的设置流程，提高可视性并降低成本。
- ❑ 在流行病期间改进对跨机构数据的访问，受到保护隐私的计算的保护。
- ❑ 提高获取和跟踪共享数据访问许可的自动化程度。
- ❑ 增强患者对隐私保护的信心。

应用过程中的注意事项：

- ❑ 通过提供隐私保护数据共享技术和新的多方计算架构的规模，可能在临床试验社区中造成积极的中断。
- ❑ 身份、身份验证和授权仍然是在支持区块链的网络上可执行的智能合约的未解决问题。
- ❑ 基于新的经济激励模型的新数据市场（例如临床试验数据市场）发展的潜在途径。

5.2　EOS 中的智能合约

相较于以太坊，EOS 系统中的智能合约承载了更多的功能，在上面的介绍中我们已经知道 EOS 采用石墨烯底层架构，通过插件与系统智能合约实现 DPoS+BFT 的共识机制以及其他必要的功能，而且 EOS 中的智能合约更具有灵活性，主要体现在以下两点：

- 节点可以通过提案的形式对系统智能合约进行升级，以达到更新系统的作用；
- 用户可以创建新的智能合约，并将其与一个已经部署上链的智能合约账户名重新绑定，以达到更新合约的效果。这里应当注意，该用户需要拥有该智能合约的更新权限，即具有这个账户名对应的公私钥。

与以太坊不同，在 EOS 系统中，每份智能合约都必须附有一份李嘉图合约，该合约定义了合同中具有法律约束力的条款和条件。李嘉图合约实质上是将现实合同向代码转化的一种具有中介性质的合约描述方式，其作用在于大幅增加了智能合约代码式条款的可读性。李嘉图合约必须真实反映智能合约代码内容，并对其进行更加具体的阐述和细节化描述，并由合约发行方进行签署。

李嘉图合约不影响智能合约代码本身的执行，但影响智能合约共识的形成。大部分人并不能轻松读懂智能合约代码所表述的内容，自然也搞不懂合约条款中存在哪些漏洞和欺骗。发布智能合约的人有动机给代码做手脚，以便于最大化自己的利益。因此 EOS 要求所有智能合约都需要有一份李嘉图合约，以便于合约接受各方的检查，加快形成共识。

1. EOS 系统的通信模型

EOS 系统中的智能合约由一组操作和类型定义组成。操作定义指定并实现合同的行为，类型定义指定所需的内容和结构。EOSIO 操作主要在基于消息的通信体系结构中运行。客户端通过发送（推送）消息来调用操作 nodeos，这可以使用 cleos 命令完成。它也可以使用 EOSIO send 方法之一（例如 eosio::action::send）完成。nodeos 将操作请求分派给实现合同的 WASM 代码。该代码完整运行，然后继续处理下一个操作。

EOSIO 智能合约可以相互通信，例如让另一个合同执行与当前交易完成相关的某些操作，或触发当前交易范围之外的未来交易。

仅供非商业用途或交流学习使用

EOSIO 支持两种基本通信模型：内联和延迟。在当前事务中执行的操作是内联操作的示例，而触发的将来事务是延迟操作的示例。

合同之间的沟通应视为异步发生。异步通信模型可能导致垃圾邮件，资源限制算法将解析垃圾邮件。

2. 内联通信

内联通信采用请求其他操作的形式，这些操作需要作为调用操作的一部分执行。内联操作使用原始事务的相同作用域和权限进行操作，并保证使用当前事务执行。这些可以有效地被认为是调用事务中的嵌套事务。如果事务的任何部分失败，则内联操作将与事务的其余部分一起展开。无论成功与否，调用内联操作都不会在事务范围之外生成任何通知。

3. 延期沟通

延迟通信在概念上采用发送到对等事务的动作通知的形式。根据生产者的判断，延迟的操作最多可以安排在稍后的时间运行，无法保证将执行延期操作。

如前所述，延迟通信将由生产者自行决定。从原始事务的角度来看，即创建延迟事务的事务，它只能确定创建请求是否已成功提交或是否失败（如果失败，则会立即失败）。延期交易具有发送它们的合同的权限。交易可以取消延期交易。

4. 交易与操作

动作表示单个操作，而事务是一个或多个动作的集合。合同和账户以行动的形式进行沟通。如果要将操作作为一个整体执行，则可以单独发送操作，也可以以组合形式发送操作。

交易只有一个动作。

```
{
    "expiration": "2018-04-01T15:20:44",
    "region": 0,
    "ref_block_num": 42580,
    "ref_block_prefix": 3987474256,
    "net_usage_words": 21,
    "kcpu_usage": 1000,
    "delay_sec": 0,
    "context_free_actions": [],
    "actions": [{
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
                "account": "eosio.Token",
                "name": "issue",
                "authorization": [{
                    "actor": "eosio",
                    "permission": "active"
                }
            ],
            "data": "00000000007015d640420f000000000004454f5300000000046d656d6f"
        }
    ],
    "signatures": [
        ""
    ],
    "context_free_data": []
}
```

具有多个操作的事务，这些操作必须全部成功，否则事务将失败。

```
{
    "expiration": "...",
    "region": 0,
    "ref_block_num": ...,
    "ref_block_prefix": ...,
    "net_usage_words": ..,
    "kcpu_usage": ..,
    "delay_sec": 0,
    "context_free_actions": [],
    "actions": [{
        "account": "...",
        "name": "...",
        "authorization": [{
                "actor": "...",
                "permission": "..."
            }
        ],
        "data": "..."
    }, {
        "account": "...",
        "name": "...",
        "authorization": [{
            "actor": "...",
            "permission": "..."
        }
    ],
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            "data": "..."
        }
    ],
    "signatures": [
        ""
    ],
    "context_free_data": []
}
```

5. 行动名称限制

动作类型实际上是 base32 编码的 64 位整数。这意味着动作类型的前 12 个字符仅能从字符 a～z、1～5 和“.”中选择。如果有第 13 个字符，则第 13 个字符仅能从“.”和 a～p 中选择。

6. 交易确认

交易完成后，将生成交易收据。此收据采用散列形式。接收事务散列并不意味着事务已被确认，它只意味着节点接受它而没有错误，这也意味着其他生产者很可能会接受它。

通过确认，你应该在事务历史记录中看到包含它的块编号的事务。

7. 操作处理程序和操作“应用”上下文

智能合约提供操作处理程序来执行所请求操作的工作。每次操作运行时，即通过 apply 在合同实现中运行操作“应用”，EOSIO 会创建一个新的操作“应用”上下文，操作在该上下文中运行。图 5-14 说明了“应用”上下文操作的关键元素。

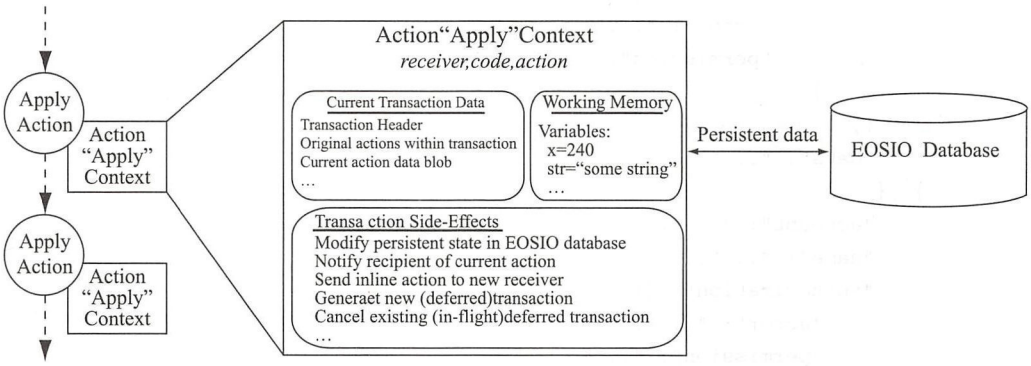


图 5-14 EOSIO 中 action 执行过程

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

从 EOSIO 区块链的全局视图来看，EOSIO 网络中的每个节点都会获得每个智能合约中的每个操作的副本并运行。一些节点正在执行智能合约的实际工作，而其他节点证明事务块的有效性。因此，重要的是合同能够确定“它们是谁”，或者确定它们在哪个环境下运行。上下文识别信息在动作上下文中提供，如图 5-14 所示的 receiver、code、action。receiver 是当前正在处理操作的账户，code 是授权合同的账户，action 是当前正在运行的操作的 ID。

如上所述，行动在交易中运作，如果事务失败，则必须回滚事务中所有操作的结果。动作上下文的关键部分是当前交易数据。这包含事务头，事务中所有原始操作的有序向量，事务中上下文自由操作的向量，由代码定义的可修复的上下文无关数据集（作为 blob 的向量提供）实现契约，以及 blob 向量的完整索引。

在处理操作之前，EOSIO 会为操作设置一个干净的工作内存，这是保持动作的工作变量的地方。操作的工作内存仅对该操作可用，即使对于同一事务中的操作也是如此。在执行另一个操作时可能已设置的变量在另一个操作的上下文中不可用。在操作之间传递状态的唯一方法是将其持久化并从 EOSIO 数据库中检索它。

动作可能有许多副作用。其中包括：

- 更改状态在 EOSIO 持久存储中保持不变。
- 通知当前交易的收件人。
- 将内联操作请求发送给新接收方。
- 生成新的（延期）交易。
- 取消现有（在飞行中）延期交易（即取消已提交的延期交易请求）。

8. 交易限制

每个事务必须在 30ms 或更短的时间内执行。如果事务包含多个操作，并且这些操作的总和大于 30ms，则整个事务将失败。在没有对其操作进行并发性要求的情况下，可以通过在单独的事务中包含 CPU 使用操作来规避这种情况。

5.3　EOS 智能合约文件结构

基于 EOSIO 的区块链使用 WebAssembly (WASM) 执行用户生成的应用程序和代码。WASM 是一项新兴的网络标准，得到了谷歌、微软、苹果等公司的广泛支持。目

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

前，用于构建编译为 WASM 的应用程序的最成熟工具链是使用 C/C++ 编译器的 clang/llvm。所以一个智能合约项目需要包含以下三种文件，以便更好地描述合约内容，且易于执行。

首先讲一下具体的开发流程：智能合约开发者首先编写 c/cpp 版本文件，并利用 eosiocpp 编译器将 cpp 文件编译成 abi、wasm 和 wast 文件，在 debug 完成后，将这三个文件统一上载至区块链中，那么每个文件的作用和结构是什么样子的呢？在下面的内容中将进行详细介绍。

5.3.1 .abi 文件

应用程序二进制接口（ABI）是一个基于 JSON 的描述，记录如何在 JSON 和二进制表示之间实现用户操作。ABI 还描述了如何将数据库状态转换为 JSON 或从 JSON 转换。通过 ABI 描述合同后，开发人员和用户将能够通过 JSON 无缝地与合同进行交互。所以可以说 abi 文件相当于该智能合约接口的描述文件，也可以说是相当于智能合约的头文件。

可以使用该 eosiocpp 工具从 .hpp 文件生成 ABI 文件。

```
$ eosiocpp -g ${contract}.abi ${contract}.hpp
```

以下是典型的 ABI 示例。

```
{
    "types": [{
        "new_type_name": "account_name",
        "type": "name"
      }
    ],
    "structs": [{
        "name": "transfer",
        "base": "",
        "fields": {
          "from": "account_name",
          "to": "account_name",
          "quantity": "uint64"
        }
      },{
        "name": "account",
        "base": "",
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            "fields": {
              "account": "name",
              "balance": "uint64"
            }
          }
        ],
        "actions": [{
            "action": "transfer",
            "type": "transfer"
          }
        ],
        "tables": [{
            "table": "account",
            "type": "account",
            "index_type": "i64",
            "key_names" : ["account"],
            "key_types" : ["name"]
          }
        ]
    }
```

该 ABI 定义了 transfer 类型。这告诉 EOS 系统，智能合约中的 transfer 函数为 action 属性，当接收到名为“transfer”的动作时，执行对应函数。

以下代码定义了一个名为 transfer 的 structs 数组。

```
    "structs": [{
        "name": "transfer",
        "base": "",
        "fields": {
            "from": "account_name",
            "to": "account_name",
            "quantity": "uint64"
        }
    },{
...
```

该 ABI 有几个参数，包括 from、to 和 quantity。这些字段具有相应的类型 account_name 和 uint64。account_name 是一个内置类型，用于表示 base32 字符串为 uint64。

5.3.2　.wasm/.wast 文件

从图 5-15 看出，webassembly 首先将 C 或 C++ 代码编译成 wasm 文件，最后结合

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

JavaScript 文件共同生成 HTML，以前端形式呈现，所以 wasm 相当于智能合约真正的执行版本。

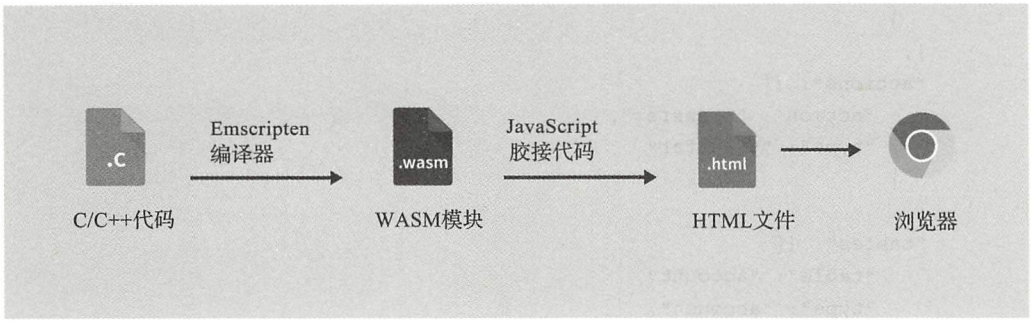


图 5-15 WebAssembly 编译流程

WebAssembly 除了定义了二进制格式以外，还定义了一份对等的文本描述 。官方给出的是线性表示的例子，而 wast 是用 S- 表达式（s-expressions）描述的另一种文本格式。利用 eosiocpp 提供的功能，我们也可以很方便地将一个 .cpp 文件翻译成 .wast 文件便于人类阅读，但请注意，这个文件并不是最终在链上执行的版本，wasm 才是。

5.3.3 eosioapp 编译器使用

eosiocpp 是 EOS 项目内置的一个将 C++ 编译为 WASM 和 ABI 文件的编译器。它生成上传到区块链的 .wasm 和 .abi 文件。为 EOS 智能合约开发人员提供方便。

使用 eosiocpp 生成 ABI 规范文件，eosiocpp 可以通过检查合约源码中声明的类型的内容来生成 ABI 规范文件。为了表明一个类型需要被导出到 ABI（以一个 action 或者一个 table 的形式），@abi 注释必须用在类型声明所附的注释中。注释的语法如下所示。

```
@abi action [name name2 ... nameN]
@abi table [index_type name]
```

为了生成 ABI 文件，eosiocpp 被调用的时候必须使用 -g 选项。

```
eosiocpp -g abi.json types.hpp
```

返回如下结果。

```
Generated abi.json ...
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

eosiocpp 也可用于生成序列化 / 反序列化 ABI 规范中定义的类型的辅助函数。

```
eosiocpp -g abi.json -gs types.hpp
```

返回如下结果。

```
Generated abi.json ...
Generated types.gen.hpp ...
```

下面介绍一个 eosiocpp 可读的智能合约示例。

声明一个 action。

```
#include <eoslib/types.hpp>
#include <eoslib/string.hpp>

//@abi action
struct action_name {
    uint64_t    param1;
    uint64_t    param2;
    eosio::string param3;
};
{
    "types": [],
    "structs": [{
        "name": "action_name",
        "base": "",
        "fields": {
            "param1": "uint64",
            "param2": "uint64",
            "param3": "string"
        }
      }
    ],
    "actions": [{
        "action_name": "actionname",
        "type": "action_name"
      }
    ],
    "tables": []
}
```

使用同样接口声明多个 action。

```
#include <eoslib/types.hpp>
#include <eoslib/string.hpp>
```

```
//@abi action action1 action2
struct action_name {
    uint64_t param1;
    uint64_t param2;
    eosio::string   param3;
};
{
    "types": [],
    "structs": [{
        "name": "action_name",
        "base": "",
        "fields": {
            "param1": "uint64",
            "param2": "uint64",
            "param3": "string"
        }
      }
    ],
    "actions": [{
        "action_name": "action1",
        "type": "action_name"
      },{
        "action_name": "action2",
        "type": "action_name"
      }
    ],
    "tables": []
}
```

声明一个 table。

```
#include <eoslib/types.hpp>
#include <eoslib/string.hpp>

//@abi table
struct my_table {
    uint64_t key;
};
{
    "types": [],
    "structs": [{
        "name": "my_table",
        "base": "",
        "fields": {
```

```
            "key": "uint64"
        }
      }
    ],
    "actions": [],
    "tables": [{
        "table_name": "mytable",
        "index_type": "i64",
        "key_names": [
            "key"
        ],
        "key_types": [
            "uint64"
        ],
        "type": "my_table"
      }
    ]
}
```

声明一个 table，带有明确的索引类型。

```
*a struct结构体中有3个uint64_t变量可以标识成 i64 或者 i64i64i64
#include <eoslib/types.hpp>

//@abi table i64
struct my_new_table {
    uint64_t key;
    uint64_t name;
    uint64_t age;
};
{
    "types": [],
    "structs": [{
        "name": "my_new_table",
        "base": "",
        "fields": {
            "key": "uint64",
            "name": "uint64",
            "age": "uint64"
        }
      }
    ],
    "actions": [],
    "tables": [{
```

```
        "table_name": "mynewtable",
        "index_type": "i64",
        "key_names": [
            "key"
        ],
        "key_types": [
            "uint64"
        ],
        "type": "my_new_table"
      }
    ]
}
```

使用同样的结构体声明一个 table 和一个 action。

```
#include <eoslib/types.hpp>
#include <eoslib/string.hpp>

/*
 * @abi table
 * @abi action
 */
struct my_type {
    eosio::string key;
    eosio::name value;
};
{
    "types": [],
    "structs": [{
        "name": "my_type",
        "base": "",
        "fields": {
            "key": "string",
            "value": "name"
        }
      }
    ],
    "actions": [{
        "action_name": "mytype",
        "type": "my_type"
      }
    ],
    "tables": [{
        "table_name": "mytype",
```

```
        "index_type": "str",
        "key_names": [
            "key"
        ],
        "key_types": [
            "string"
        ],
        "type": "my_type"
      }
    ]
}
```

定义导出的例子。

```
#include <eoslib/types.hpp>
struct simple {
    uint64_t u64;
};

typedef simple simple_alias;
typedef eosio::name name_alias;

//@abi action
struct action_one : simple_alias {
    uint32_t u32;
    name_alias name;
};
{
    "types": [{
        "new_type_name": "simple_alias",
        "type": "simple"
      },{
        "new_type_name": "name_alias",
        "type": "name"
      }
    ],
    "structs": [{
        "name": "simple",
        "base": "",
        "fields": {
            "u64": "uint64"
        }
      },{
        "name": "action_one",
```

```
        "base": "simple_alias",
        "fields": {
            "u32": "uint32",
            "name": "name_alias"
        }
      }
    ],
    "actions": [{
        "action_name": "actionone",
        "type": "action_one"
      }
    ],
    "tables": []
}
```

使用生成的序列化 / 反序列化函数。

```
#include <eoslib/types.hpp>
#include <eoslib/string.hpp>

struct simple {
    uint32_t u32;
    fixed_string16 s16;
};

struct my_complex_type {
    uint64_t u64;
    eosio::string str;
    simple simple;
    bytes bytes;
    public_key pub;
};

typedef my_complex_type complex;

//@abi action
struct test_action {
    uint32_t u32;
    complex cplx;
};
void apply( uint64_t code, uint64_t action ) {
    if( code == N(mycontract) ) {
        if( action == N(testaction) ) {
            auto msg = eosio::current_message<test_action>();
            eosio::print("test_action content\n");
```

```
                eosio::dump(msg);

                bytes b = eosio::raw::pack(msg);
                printhex(b.data, b.len);
            }
        }
    }
```

5.4　EOS 智能合约依赖库介绍

经过上面的介绍，读者应该对 EOS 中的智能合约有了一定的理解，为了方便开发人员，我们在这里将 EOS 系统中的合约依赖库中的一些常用 API 功能与用法罗列一下。

1. Action 文件

这部分定义了查询或发送 action 的 API。在 EOS 系统中，一个 action 包括 action 的作用域、action 名称、授权账户与授权等级以及具体的执行代码，结构如下所示。

```
struct action {
    scope_name scope; // 该action的作用域
    action_name name; // 该action的名称
    permission_level[] authorization; //该action的授权信息（账户与权限等级）
    bytes data; // 具体待执行数据
};
```

Action 部分中成员函数以及功能描述如表 5-1 所示。

表 5-1　Action 成员函数以及功能描述

成员函数	描　述
public uint32_t read_action_data (void * msg,uint32_t len)	拷贝当前 action 数据到指定位置，其中 msg 指针为待拷贝位置，len 为待拷贝数据长度
public uint32_t action_data_size()	获得当前 action 数据长度
public void require_recipient(account_namename)	将指定账户添加到通知列表中
public void require_auth(account_namename)	判断指定账户是否位于授权列表中
public bool has_auth(account_namename)	判断当前 action 中是否有指定账户的授权信息，返回 bool 值
public void require_auth2(account_name name, permission_name permission)	判断指定账户与权限等级是否位于授权列表中

（续）

成员函数	描　述
public bool is_account(account_name name)	判断账户名合法性
public void send_inline(char * serialized_action,size_t size)	在 action 的父交易中的上下文中，发送一个内联 action
public void send_context_free_inline(char * serialized_action,size_t size)	在 action 的父交易中的上下文中，发送一个上下文自由的内联 action
public void require_write_lock(account_namename)	判断指定账户是否位于写入锁定列表中
public void require_read_lock(account_namename)	判断指定账户是否位于读取锁定列表中
public uint64_t publication_time()	获取公共时间
Public account_name`[current_receiver](#current_receiver) ()	获取当前 action 的接收者

2. Chain 文件

这部分定义了查询当前区块链状态信息，成员函数以及功能描述如表 5-2 所示。

表 5-2　Chain 成员函数以及功能描述

成员函数	函数描述
public uint32_tget_active_producers (account_name* producers, uint32_t datalen)	获取活跃出块节点账户集

其中参数包括两个，producers 是一个用于保存返回账户名的 buffer 指针，datalen 为返回的数据长度。该函数包含一个前置条件，那就是 producers 所指向的内存空间必须不小于 datalen，执行成功后，函数将返回的账户信息注入 producers 指向的 buffer 中。

3. Console 文件

这部分定义了一些打印文本信息的方法，成员函数以及功能描述如表 5-3 所示。

表 5-3　Console 成员函数以及功能描述

成员函数	函数描述
public void prints(const char * cstr)	打印字符串变量
public void prints_l(const char * cstr,uint32_t len)	打印指定长度的字符串变
public void printi(int64_t value)	打印一个 64 位签名整数
public void printui(uint64_t value)	打印一个 64 位非签名整数

非卖品，仅供非商业用途或交流学习使用

（续）

成员函数	函数描述
public void printi128(const int128_t * value)	打印一个 128 位签名整数
public void printui128(const uint128_t * value)	打印一个 128 位非签名整数
public void printsf(float value)	打印一个单精度浮点数
public void printdf(double value)	打印一个双精度浮点数
public void printqf(const long double * value)	打印值为四倍精度浮点数（即长双精度）
public void printn(uint64_t name)	打印 64 位名称作为 base32 编码的字符串
public void printhex(const void * data,uint32_t datalen)	打印 64 位名称作为 base32 编码的字符串

4. Crypto

该文件定义了一个计算与验证哈希值的 C 函数，成员函数以及功能描述如表 5-4 所示。

表 5-4　Crypto 成员函数以及功能描述

成员函数	函数描述
public void assert_sha256(char * data,uint32_t length,const checksum256 * hash)	测试从数据生成的 sha256 哈希是否与提供的校验和匹配
public voidassert_sha1(char * data,uint32_t length,const checksum160 * hash)	测试从数据生成的 sha1 哈希是否与提供的校验和匹配
public voidassert_sha512(char * data,uint32_t length,const checksum512 * hash)	测试从数据生成的 sha512 哈希是否与提供的校验和匹配
public voidassert_ripemd160(char * data,uint32_t length,const checksum160 * hash)	测试从数据生成的 ripemod160 哈希是否与提供的校验和匹配
public voidsha256(char * data,uint32_t length,checksum256 * hash)	使用 sha256 散列数据并将结果存储在散列指向的内存中
public voidsha1(char * data,uint32_t length,checksum160 * hash)	使用 sha1 散列数据并将结果存储在散列指向的内存中
public voidsha512(char * data,uint32_t length,checksum512 * hash)	使用 sha512 散列数据并将结果存储在散列指向的内存中
public voidripemd160(char * data,uint32_t length,checksum160 * hash)	使用 ripemod160 散列数据并将结果存储在散列指向的内存中
public intrecover_key(const checksum256 * digest,const char * sig,size_t siglen,char * pub,size_t publen)	计算用于给定签名的公钥和用于创建消息的哈希
public voidassert_recover_key(const checksum256 * digest,const char * sig,size_t siglen,const char * pub,size_t publen)	使用摘要和签名生成的密钥测试给定的公钥

非卖品，仅供非商业用途或交流学习使用

5. Database

该文件定义了与数据库交互的 C 函数，成员函数以及功能描述如表 5-5 所示。

表 5-5 Database 成员函数以及功能描述

成员函数	函数描述
public int32_tdb_store_i64(account_namescope,table_nametable,account_namepayer,uint64_t id,const void * data,uint32_t len)	将记录存储在主 64 位整数索引表中
public voiddb_update_i64(int32_t iterator,account_namepayer,const void * data,uint32_t len)	更新主 64 位整数索引表中的记录
public voiddb_remove_i64(int32_t iterator)	删除主 64 位整数索引表中的记录
public int32_tdb_get_i64(int32_t iterator,const void * data,uint32_t len)	获取主 64 位整数索引表中的记录
public int32_tdb_next_i64(int32_t iterator,uint64_t * primary)	获取主 64 位整数索引表中的记录
public int32_tdb_previous_i64(int32_t iterator,uint64_t * primary)	从主 64 位整数索引表中获取给定迭代器之前的上一条记录
public int32_tdb_find_i64(account_namecode,account_namescope,table_nametable,uint64_t id)	在主 64 位整数索引表中查找记录
public int32_tdb_lowerbound_i64(account_namecode,account_namescope,table_nametable,uint64_t id)	给定主 64 位整数索引表中的键的下限记录
public int32_tdb_upperbound_i64(account_namecode,account_namescope,table_nametable,uint64_t id)	给定主 64 位整数索引表中的键的上限记录
public int32_tdb_end_i64(account_namecode,account_namescope,table_nametable)	在主 64 位整数索引表中查找最新记录
public int32_tdb_idx64_store(account_namescope,table_nametable,account_namepayer,uint64_t id,const uint64_t * secondary)	将记录的辅助索引存储在辅助 64 位整数索引表中
public voiddb_idx64_update(int32_t iterator,account_namepayer,const uint64_t * secondary)	在辅助 64 位整数索引表中更新记录的辅助索引
public voiddb_idx64_remove(int32_t iterator)	从辅助 64 位整数索引表中删除记录的辅助索引
public int32_tdb_idx64_next(int32_t iterator,uint64_t * primary)	获取辅助 64 位整数索引表中的下一个辅助索引
public int32_tdb_idx64_previous(int32_t iterator,uint64_t * primary)	在辅助 64 位整数索引表中获取先前的辅助索引
public int32_tdb_idx64_find_primary(account_namecode,account_namescope,table_nametable,uint64_t * secondary,uint64_t primary)	在给定记录的主键的情况下，从辅助 64 位整数索引表中获取记录的二级索引

非卖品，仅供非商业用途或交流学习使用

（续）

成员函数	函数描述
public int32_tdb_idx64_find_secondary(account_namecode,account_namescope,table_nametable,const uint64_t * secondary,uint64_t * primary)	在给定辅助索引键的情况下，从辅助 64 位整数索引表中获取记录的二级索引
public int32_tdb_idx64_lowerbound(account_namecode,account_namescope,table_nametable,uint64_t * secondary,uint64_t * primary)	在给定辅助索引键的情况下，从辅助 64 位整数索引表中获取记录的二级索引
public int32_tdb_idx64_upperbound(account_namecode,account_namescope,table_nametable,uint64_t * secondary,uint64_t * primary)	在给定辅助索引键的情况下，从辅助 64 位整数索引表中获取记录的二级索引
public int32_tdb_idx64_end(account_namecode,account_namescope,table_nametable)	从辅助 64 位整数索引表中获取最后一个辅助索引

后面函数名雷同的都具有相同功能，只不过索引值长度不同，所以在这里不做重复介绍，需要强调的是它们的参数：

- scope- 存储二级索引的范围。
- table- 存储二级索引的表名。
- payer- 为此存储空间付款的账户。
- id- 要存储的二级索引的记录的主键。
- secondary- 指向要存储的辅助索引的键的指针。
- iterator- 二级索引的迭代器。

6. System 文件

该文件定义了用于与系统级内在函数交互的 API，成员函数以及功能描述如表 5-6 所示。

表 5-6　System 成员函数以及功能描述

成员函数	函数描述
public voideosio_assert(uint32_t test,const char * msg)	中止处理此操作并展开所有挂起的更改
public voideosio_assert_message(uint32_t test,const char * msg,uint32_t msg_len)	中止处理此操作并展开所有挂起的更改
public voideosio_assert_code(uint32_t test,uint64_t code)	中止处理此操作并展开所有挂起的更改
public voideosio_exit(int32_t code)	在不失败合同的情况下中止执行 wasm
public uint64_tcurrent_time()	获取最后接受的块的时间
public uint32_tnow()	获取当前块的时间（向下舍入到最接近的秒）（即包含此操作的块）

非卖品，仅供非商业用途或交流学习使用

7. Transaction 文件

该文件定义了发送交易的 API，成员函数以及功能描述如表 5-7 所示。

表 5-7 Transaction 成员函数以及功能描述

成员函数	函数描述
public voidsend_deferred(const uint128_t & sender_id,account_namepayer,const char * serialized_transaction,size_t size,uint32_t replace_existing)	发送延期交易
public intcancel_deferred(const uint128_t & sender_id)	取消延期交易
public size_tread_transaction(char * buffer,size_t size)	访问当前正在执行的事务的副本
public size_ttransaction_size()	获取当前正在执行的事务的大小
public inttaPoS_block_num()	获取当前正在执行的事务上用于 TAPoS 的块编号
public inttaPoS_block_prefix()	获取当前正在执行的事务上用于 TAPoS 的块前缀
publictime`[expiration](#expiration)()`	获取当前正在执行的事务的到期时间
public intget_action(uint32_t type,uint32_t index,char * buff,size_t size)	从活动事务中检索指示的操作
public intget_context_free_data(uint32_t index,char * buff,size_t size)	检索被签名的交易数据

总结一些关键参数的含义如下所示。

- sender_id：发件人的 ID。
- payer：账户支付 RAM。
- serialized_transaction ：要延迟的序列化交易的指针。
- size ：要保留的数据大小。

8. Privileged 文件

该文件定义了优先级 API，成员函数以及功能描述如表 5-8 所示。

表 5-8 Privileged 成员函数以及功能描述

成员函数	函数描述
public voidset_resource_limits(account_nameaccount,int64_t ram_bytes,int64_t net_weight,int64_t cpu_weight)	设置账户的资源限制
public int64_tset_proPoSed_producers(char * producer_data,uint32_t producer_data_size)	设置新的活跃区块生产者提案
public voidset_active_producers(char * producer_data,uint32_t producer_data_size)	设置新的活动生成器，只有在下一轮开始的区块不可逆时，生产者才会被激活

非卖品，仅供非商业用途或交流学习使用

（续）

成员函数	函数描述
public boolis_privileged(account_nameaccount)	检查账户是否具有特权
public voidset_privileged(account_nameaccount,bool is_priv)	设置账户的特权状态
public voidset_blockchain_parameters_packed(char * data,uint32_t datalen)	设置区块链关键参数
public uint32_tget_blockchain_parameters_packed(char * data,uint32_t datalen)	获取区块链的关键参数
public voidactivate_feature(int64_t f)	激活新的特征

其中一些关键参数如下所示。

- account- 要设置其资源限制的账户的名称。
- ram_bytes-ram 限制值。
- net_weight- 净限额。
- cpu_weight-cpu 限制值。
- set_proPoSed_producers- 提出新的活跃生产者计划。
- producer_data- 以适当的生产者计划顺序打包的 produce_keys 数据。
- producer_data_size- 数据缓冲区的大小。

9. Memory 文件

该文件定义了一些分配内存的 C 函数，成员函数以及功能描述如表 5-9 所示。

表 5-9　Memory 成员函数以及功能描述

成员函数	函数描述
public void *malloc(size_t size)	分配额外的内存
public void *calloc(size_t count,size_t size)	为计数元素数组分配一块内存，每个计数元素的长度大小为字节，并用 0 初始化所有位
public void *realloc(void * ptr,size_t size)	重新分配给定的内存区域
public voidfree(void * ptr)	释放给定的内存区域

定义类型安全的 C++ 包装器，用于查询操作和发送操作。这些包装好的文件通常以宏的形式调用 C 文件中的 API，使函数功能更加直观，下面我们选择其中最为关键的部分——Multi-Index 做详细介绍。

非卖品，仅供非商业用途或交流学习使用

EOSIO Multi-Index API 为 EOSIO 数据库提供 C++ 接口。它是在 Boost Multi Index Container 的基础之上开发而成。EOSIO Multi-Index 表只需要一个 uint64_t 主键。为了使表能够检索主键，存储在表中的对象需要具有名为 primary_key() 的 const 成员函数，该函数返回 uint64_t。EOSIO Multi-Index 表还支持最多 16 个二级索引。二级指数的类型可以是以下任何一种。

- uint64_t
- uint128_t
- uint256_t
- double
- long double

它的成员函数以及功能描述如表 5-10 所示。

表 5-10 Multi-Index 成员函数以及功能描述

成员函数	函数描述
public inlinemulti_index(uint64_t code,uint64_t scope)	通过主迭代器加载对象
public inline uint64_t get_code() const	返回代码成员属性
public inline uint64_tget_scope() const	返回范围成员属性
public inline const_iteratorcbegin() const	返回指向具有 Multi-Index 表中最低主键值的 object_type 的迭代器
public inline const_iteratorbegin() const	返回指向具有 Multi-Index 表中最低主键值的 object_type 的迭代器
public inline const_iteratorcend() const	返回指向具有 Multi-Index 表中最高主键值的 object_type 的迭代器
public inline const_iterator end() const	返回指向具有 Multi-Index 表中最高主键值的 object_type 的迭代器
public inlineconst_reverse_iterator crbegin() const	返回指向具有 Multi-Index 表中最高主键值的 object_type 的反向迭代器
public inlineconst_reverse_iterator rbegin() const	返回指向具有 Multi-Index 表中最高主键值的 object_type 的反向迭代器
public inlineconst_reverse_iterator crend() const	返回指向具有 Multi-Index 表中最低主键值的 object_type 的迭代器
public inlineconst_reverse_iterator [rend](#rend)() const`	返回指向具有 Multi-Index 表中最低主键值的 object_type 的迭代器
public inline const_iteratorlower_bound (uint64_t primary) const	搜索具有大于或等于给定主键的最低主键的 object_type。（下界元素）

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

（续）

成员函数	函数描述
public inline const_iteratorupper_bound (uint64_t primary) const	搜索具有大于给定主键的最低主键的 object_type。（上界元素）
public inline uint64_tavailable_primary_key() const	返回可用的主键
public template<> inline autoget_index()	返回适当类型的二级索引
public template<> inline autoget_index() const	返回适当类型的二级索引
public inline const_iteratoriterator_to(const T & obj) const	返回 Multi-Index 表中给定对象的迭代器
public template<> inlineconst_iterator`[emplace](#emplace)(uint64_t payer,Lambda && constructor)`	向表中添加新对象（即行）
public template<> inline voidmodify(const_iteratoritr,uint64_t payer,Lambda && updater)	修改表中的现有对象
public template<> inline voidmodify(const T & obj,uint64_t payer,Lambda && updater)	修改表中的现有对象
public inline const T &get(uint64_t primary,const char * error_msg) const	使用主键从表中检索现有对象
public inlineconst_iterator`[find](#find)(uint64_t primary) const`	使用主键在表中搜索现有对象
public inlineconst_iterator`[erase](#erase)([const_iterator](#const_iterator) itr)`	使用主键从表中删除现有对象
public inline void erase(const T & obj)	使用主键从表中删除现有对象
typedef const_reverse_iterator	通过宏方法定义 multi_index 迭代器

其中的关键参数如下所示。

- code- 拥有该永久表的账户。
- scope- 该表中元素的作用域，即该对象的所有者。

先验条件：代码和范围成员属性已初始化。

后验条件：

- 每个二级索引表都已初始化。
- 更新辅助索引以引用新添加的对象。如果辅助索引表不存在，则创建它们。
- 当使用者储存超过原表存储空间的数据时，将自动创建新的存储空间，并且支付创建所需的开销，如 RAM 等。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

10. get_code 函数

该函数用于返回 code 合约的成员属性。

返回值：拥有该表的智能合约账户名。

示例：

```
#include
using namespace eosio;
using namespace std;
class addressbook: contract {
//声明address结构对象
    struct address {
        uint64_t account_name;
        string first_name;
        string last_name;
        string street;
        string city;
        string state;
        uint64_t primary_key() const { return account_name; }
        EOSLIB_SERIALIZE( address, (account_name)(first_name)(last_name)
            (street)(city)(state) )
    };
    public:
//构造函数
        addressbook(account_name self):contract(self) {}
typedef eosio::multi_index< N(address), address > address_index;
//声明multi_index表用于存储address对象，并用address_index作为表别名
void myaction() {
//初始化表，其中前一个参数为拥有该表的合约名称，后一个参数为该表作用
        address_index addresses(N(dan), N(dan));
//返回addresses表的合约名，并判断是否等于dan，若不相等则报Codes does' t match错误
        eosio_assert(addresses.get_code() == N(dan), "Codes don't match.");
    }
}
EOSIO_ABI( addressbook, (myaction) )
```

11. get_scope 函数

该函数用于返回当前接收者代码中作用域标识，可通过这个标识找到主表实例。

示例：

```
#include
using namespace eosio;
using namespace std;
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
class addressbook: contract {
//定义address数据结构，包括账户名、真实姓名、街道、城市等详细记录
struct address {
    uint64_t account_name;
    string first_name;
    string last_name;
    string street;
    string city;
    string state;
    uint64_t primary_key() const { return account_name; }
    EOSLIB_SERIALIZE( address, (account_name)(first_name)(last_name)(street)
        (city)(state) )
};
public:
    addressbook(account_name self):contract(self) {}
    typedef eosio::multi_index< N(address), address > address_index;
    void myaction() {
        address_index addresses(N(dan), N(dan));
//返回addresses表的作用域，并判断是否等于dan，若不相等则报Scopes don't match错误
        eosio_assert(addresses.get_scope() == N(dan), "Scopes don't match");
    }
}
EOSIO_ABI( addressbook, (myaction) )
```

12. Cbegin/begin/crbegin/rbegin 函数

该函数用于返回指向具有 Multi-Index 表中第一个主键值的 object_type 的迭代器（以 r 为前缀的函数返回的是反向迭代器，即从末尾到开头）。

示例：

```
#include
using namespace eosio;
using namespace std;
class addressbook: contract {
    struct address {
        uint64_t account_name;
        string first_name;
        string last_name;
        string street;
        string city;
        string state;
        uint64_t primary_key() const { return account_name; }
        EOSLIB_SERIALIZE( address, (account_name)(first_name)(last_name)
            (street)(city)(state) )
```

仅供非商业用途或交流学习使用

```
    };
    public:
        addressbook(account_name self):contract(self) {}
        typedef eosio::multi_index< N(address), address > address_index;
        void myaction() {
            address_index addresses(_self, _self);  // code, scope
            //将指定数据添加到表中，其中_self为用于支付存储空间费用的账户，第二个是利用
              匿名函数初始化待压入对象
            addresses.emplace(_self, [&](auto& address) {
                address.account_name = N(dan);
                address.first_name = "Daniel";
                address.last_name = "Larimer";
                address.street = "1 EOS Way";
                address.city = "Blacksburg";
                address.state = "VA";
            });
//声明一个用于在表中检索dan对象的指针
            auto itr = addresses.find(N(dan));
//判断上面的指针是否位于该表首位，若不是，则报"Only address is not at front."错误
            eosio_assert(itr == addresses.cbegin(), "Only address is not at
                front.");
        }
}
EOSIO_ABI( addressbook, (myaction) )
```

13. Cend/end/Crend/rend 函数

该函数用于返回指向具有 Multi-Index 表中最后一个主键值的 object_type 的迭代器。主要用于判断某对象是否存在于表中。

示例：

```
#include
using namespace eosio;
using namespace std;
class addressbook: contract {
    struct address {
        uint64_t account_name;
        string first_name;
        string last_name;
        string street;
        string city;
        string state;
        uint64_t primary_key() const { return account_name; }
        EOSLIB_SERIALIZE( address, (account_name)(first_name)(last_name)
            (street)(city)(state) )
```

仅供非商业用途或交流学习使用

```
    };
    public:
        addressbook(account_name self):contract(self) {}
        typedef eosio::multi_index< N(address), address > address_index;
        void myaction() {
            address_index addresses(_self, _self); // code, scope
            // add to table, first argument is account to bill for storage
            addresses.emplace(_self, [&](auto& address) {
                address.account_name = N(dan);
                address.first_name = "Daniel";
                address.last_name = "Larimer";
                address.street = "1 EOS Way";
                address.city = "Blacksburg";
                address.state = "VA";
            });
//声明一个用于在表中检索dan对象的指针
            auto itr = addresses.find(N(dan));
//判断上面的指针是否位于该表末尾，若不是，则报" Address for account doesn't exist "错误
            eosio_assert(itr != addresses.cend(), "Address for account
                doesn't exist");
        }
}
EOSIO_ABI( addressbook, (myaction) )
```

14. Lower/upper_bound 函数

该函数主要用于表的检索，具体作用可以参见 std 返回一个指向 object_type 具有大于或等于 / 大于的最小主键的迭代器 primary。如果找不到对象，它将返回 end 迭代器。如果表不存在，将返回 -1。直白点说，这个函数就是用来在一个有序表中检索指定元素的下界 / 上界。

参数 primary 为下界 / 上界搜索建立目标值的主键。

示例：

```
#include
using namespace eosio;
using namespace std;
class addressbook: contract {
    struct address {
        uint64_t account_name;
        string first_name;
        string last_name;
        string street;
```

仅供非商业用途或交流学习使用

```
        string city;
        string state;
        uint32_t zip = 0;
        uint64_t primary_key() const { return account_name; }
        uint64_t by_zip() const { return zip; }
        EOSLIB_SERIALIZE( address, (account_name)(first_name)(last_name)
            (street)(city)(state)(zip) )
    };
    public:
        addressbook(account_name self):contract(self) {}
        typedef eosio::multi_index< N(address), address, indexed_by< N(zip),
            const_mem_fun > address_index;
        void myaction() {
            address_index addresses(_self, _self);  // code, scope
            addresses.emplace(payer, [&](auto& address) {
                address.account_name = N(dan);
                address.first_name = "Daniel";
                address.last_name = "Larimer";
                address.street = "1 EOS Way";
                address.city = "Blacksburg";
                address.state = "VA";
                address.zip = 93446;
            });
            addresses.emplace(payer, [&](auto& address) {
                address.account_name = N(brendan);
                address.first_name = "Brendan";
                address.last_name = "Blumer";
                address.street = "1 EOS Way";
                address.city = "Hong Kong";
                address.state = "HK";
                address.zip = 93445;
            });
            uint32_t zipnumb = 93445;
            auto zip_index = addresses.get_index();
            auto itr = zip_index.lower_bound(zipnumb);
            eosio_assert(itr->account_name == N(brendan), "Incorrect First
                Lower Bound Record ");
            itr++;
            eosio_assert(itr->account_name == N(dan), "Incorrect Second Lower
                Bound Record");
            itr++;
            eosio_assert(itr == zip_index.end(), "Incorrect End of Iterator");
    }
}
EOSIO_ABI( addressbook, (myaction) )
```

仅供非商业用途或交流学习使用

15. Emplace 函数

该函数用于在表中添加新的对象。

主要参数如下所示。

- payer：购买存储空间以保存新对象的账户名称。
- constructor：ambda 函数，用于对要在表中创建的对象进行就地初始化 Precondition。

执行过程：

- 在 MultiIndex 表中创建一个新对象，该对象具有唯一的主键（在对象中指定）。对象被序列化并写入表中。如果该表不存在，则创建该表。
- 更新辅助索引以引用新添加的对象。如果辅助索引表不存在，则创建它们。
- 创建者负责提供新对象的存储所需的开销，如果必须创建表（和二级索引表），则为表创建的开销。

示例：

```
#include
using namespace eosio;
using namespace std;
class addressbook: contract {
    struct address {
        uint64_t account_name;
        string first_name;
        string last_name;
        string street;
        string city;
        string state;
        uint64_t primary_key() const { return account_name; }
        EOSLIB_SERIALIZE( address, (account_name)(first_name)(last_name)
            (street)(city)(state) )
    };
    public:
        addressbook(account_name self):contract(self) {}
        typedef eosio::multi_index< N(address), address > address_index;
        void myaction() {
            address_index addresses(_self, _self); // code, scope
//将指定数据添加到表中，其中_self为用于支付存储空间费用的账户，第二个是利用匿名函数初始化待
  压入对象
            addresses.emplace(_self, [&](auto& address) {
                address.account_name = N(dan);
                address.first_name = "Daniel";
```

仅供非商业用途或交流学习使用

```
                address.last_name = "Larimer";
                address.street = "1 EOS Way";
                address.city = "Blacksburg";
                address.state = "VA";
            });
        }
}
EOSIO_ABI( addressbook, (myaction) )
```

16. Modify 函数

该函数用于更改一个已经存在的目标。

主要参数如下所示。

- itr：指向要更新的对象的迭代器。
- payer：更新行的存储使用情况的付款人的账户名称。
- updater：更新目标对象的 lambda 函数。

先验条件：

- Itr 必须指向现有元素。
- payer 是一个有效执行操作的有效账户，需要为存储使用付费。

后验条件：

- 已修改的对象已序列化，然后替换表中的现有对象。
- 二级指数更新，更新对象的主键不会更改。
- 付款人需要为更新对象的存储使用付费。
- 如果付款人与现有付款人相同，则付款人仅支付现有和更新对象之间的使用差异（如果此差异为负，则退款）。
- 如果付款人与现有付款人不同，则现有付款人将退还现有对象的存储使用情况。

示例：

```
#include
using namespace eosio;
using namespace std;
class addressbook: contract {
    struct address {
        uint64_t account_name;
        string first_name;
        string last_name;
        string street;
```

```
        string city;
        string state;
        uint64_t primary_key() const { return account_name; }
        EOSLIB_SERIALIZE( address, (account_name)(first_name)(last_name)
            (street)(city)(state) )
    };
    public:
        addressbook(account_name self):contract(self) {}
        typedef eosio::multi_index< N(address), address > address_index;
        void myaction() {
            address_index addresses(_self, _self); // code, scope
            // add to table, first argument is account to bill for storage
            addresses.emplace(_self, [&](auto& address) {
                address.account_name = N(dan);
                address.first_name = "Daniel";
                address.last_name = "Larimer";
                address.street = "1 EOS Way";
                address.city = "Blacksburg";
                address.state = "VA";
            });
            auto itr = addresses.find(N(dan));
            eosio_assert(itr != addresses.end(), "Address for account not found");
//修改表中主键为dan的对象属性
            addresses.modify( itr, account payer, [&]( auto& address ) {
                address.city = "San Luis Obispo";
                address.state = "CA";
            });
        }
}
EOSIO_ABI( addressbook, (myaction) )
```

17. get

该函数用于根据主键值检索并返回表中已存在的对象。

主要参数：primary 为待检索对象对应的主键值。

示例用法：

```
#include
using namespace eosio;
using namespace std;
class addressbook: contract {
    struct address {
        uint64_t account_name;
        string first_name;
        string last_name;
```

```
        string street;
        string city;
        string state;
        uint64_t primary_key() const { return account_name; }
        EOSLIB_SERIALIZE( address, (account_name)(first_name)(last_name)
            (street)(city)(state) )
    };
    public:
        addressbook(account_name self):contract(self) {}
        typedef eosio::multi_index< N(address), address > address_index;
        void myaction() {
            address_index addresses(_self, _self); // code, scope
            // add to table, first argument is account to bill for storage
            addresses.emplace(_self, [&](auto& address) {
                address.account_name = N(dan);
                address.first_name = "Daniel";
                address.last_name = "Larimer";
                address.street = "1 EOS Way";
                address.city = "Blacksburg";
                address.state = "VA";
            });
            auto user = addresses.get(N(dan)); //在表中检索并获取主键值为dan的对象
            eosio_assert(user.first_name == "Daniel", "Couldn't get him.");
        }
}
EOSIO_ABI( addressbook, (myaction) )
```

同样的用法还有 find、erase 函数，其中 find 用于检索并返回指定键值对应对象的迭代器，erase 函数用于删除表中对应对象，用法与 modify 相似，所以不做展开介绍。那么接下来，我们将会结合三个典型的案例，手把手带你开发智能合约。

5.5 EOS 智能合约实战项目

本节将会介绍两个典型的智能合约的详细开发过程，手把手带你完成智能合约的开发上链全流程。

5.5.1 HelloWorld 合约

我们现在将创建我们的第一个“hello world”合同。创建一个名为“hello”的新文件夹，cd 进入该文件夹，然后创建一个文件“hello.cpp”，其中包含以下内容。

```
HELLO.CPP
#include <eosiolib/eosio.hpp>
#include <eosiolib/print.hpp>
using namespace eosio;

class hello : public eosio::contract {
    public:
        using contract::contract;

        /// @abi action
//这个标识用于告诉编译器下面定义的函数需要包含在abi文件中，供其他节点调用
        void hi( account_name user ) {
//打印hello+用户名信息
            print( "Hello, ", name{user} );
        }
};

EOSIO_ABI( hello, (hi) )
```

你可以将代码编译为 wast 文件，以便更好地描述每个函数的功能，如下所示。

```
$ eosiocpp -o hello.wast hello.cpp
```

注意：编译器可能会生成警告。这些可以安全地忽略。

现在生成 abi。

```
$ eosiocpp -g hello.abi hello.cpp
Generated hello.abi
```

创建一个账户并上传合同。

```
$ cleos create account eosio hello.code EOS7ijWCBmoXBi3CgtK7DJxentZZeTkeUnaS
     Dvyro9dq7Sd1C3dC4 EOS7ijWCBmoXBi3CgtK7DJxentZZeTkeUnaSDvyro9dq7Sd1C3dC4
...
$ cleos set contract hello.code ../hello -p hello.code@active
...
```

现在我们可以运行部署好的智能合约。

```
$ cleos push action hello.code hi '["user"]' -p user@active
executed transaction: 4c10c1426c16b1656e802f3302677594731b380b18a44851d38e
    8b5275072857  244 bytes  1000 cycles
#    hello.code <= hello.code::hi                {"user":"user"}
>> Hello, user
```

如果你正在观察输出 nodeos，很可能看不到任何输出结果。你可以 nodeos 使用 --contracts-console 选项重新启动，以使打印的调试输出发送到控制台。输出如下所示。

```
1025500ms thread-0   producer_plugin.cpp:944         produce_block          ]
    Produced block 00004de945a23f63... #19945 @ 2018-05-25T19:17:05.500
    signed by eosio [trxs: 0, lib: 19944, confirmed: 0]
1025830ms thread-0   apply_context.cpp:28           print_debug            ]
[(hello.code,hi)->hello.code]: CONSOLE OUTPUT BEGIN =====================
Hello, user
[(hello.code,hi)->hello.code]: CONSOLE OUTPUT END   =====================
1026000ms thread-0   producer_plugin.cpp:944         produce_block          ]
    Produced block 00004deaebee2dc5... #19946 @ 2018-05-25T19:17:06.000
    signed by eosio [trxs: 1, lib: 19945, confirmed: 0]
```

此时合约允许任何人授权，我们也可以这样操作。

```
$ cleos push action hello.code hi '["user"]' -p tester@active
executed transaction: 28d92256c8ffd8b0255be324e4596b7c745f50f85722d0c4400471
    bc184b9a16  244 bytes  1000 cycles
#    hello.code <= hello.code::hi                {"user":"user"}
>> Hello, user
```

在这种情况下，tester 是授权它的人，而 user 只是一个参数。如果我们希望调用我们的合约中的 action 时进行身份验证，只有 user 账户可以调用并返回结果，那么我们需要修改合同以要求身份验证。

修改 hello.cpp 中的 hi() 函数，如下所示。

```
void hi( account_name user ) {
    require_auth( user );
    print( "Hello, ", name{user} );
}
```

重复这些步骤以编译 wast 文件并生成 abi，然后再次设置合同以部署更新。

现在，如果我们尝试使用户和权限不匹配，合同将抛出错误。

```
$ cleos push action hello.code hi '["tester"]' -p user@active
Error 3030001: missing required authority
Ensure that you have the related authority inside your transaction!;
If you are currently using 'cleos push action' command, try to add the
     relevant authority using -p option.
Error Details:
missing authority of tester
```

非卖品，仅供非商业用途或交流学习使用

调试输出结果如下。

```
1340500ms thread-0   producer_plugin.cpp:944         produce_block          ]
    Produced block 0000505f9de7f4b9... #20575 @ 2018-05-25T19:22:20.500
    signed by eosio [trxs: 0, lib: 20574, confirmed: 0]
1340515ms thread-0   http_plugin.cpp:369           handle_exception      ] FC
    Exception encountered while processing chain.push_transaction: 3090004
    missing_auth_exception: missing required authority
missing authority of user
    {"account":"user"}
    thread-0  apply_context.cpp:123 require_authorization

    {"_pending_console_output.str()":""}
    thread-0  apply_context.cpp:53 exec_one
1341005ms thread-0   producer_plugin.cpp:944         produce_block          ]
    Produced block 000050601ffc5bd9... #20576 @ 2018-05-25T19:22:21.000
    signed by eosio [trxs: 0, lib: 20575, confirmed: 0]
```

我们可以通过给予测试者调用权限来解决这个问题。

```
$ cleos push action hello.code hi '["tester"]' -p tester@active
executed transaction: 235bd766c2097f4a698cfb948eb2e709532df8d18458b92c9c6aae
    74ed8e4518  244 bytes  1000 cycles
#    hello.code <= hello.code::hi                {"user":"tester"}
>> Hello, tester
```

每份智能合约都必须有匹配的李嘉图合约。李嘉图合约规定了与智能合约的每个行动相关的具有法律约束力的行为，通过 .md 文件编写，最后将会随合约文件一并上链，而后所有调用合约 action 的 transaction 都将对这些人类可读且具有法律约束力的文字进行签名，责任可追溯。此处列出了 Hello World 合同的李嘉图合同。

```
## CONTRACT FOR HELLO WORLD
### Parameters
Input parameters: NONE

Implied parameters:

* _**account_name**_ (name of the party invoking and signing the contract)

### Intent
INTENT. The intention of the author and the invoker of this contract is to
    print output. It shall have no other effect.

### Term
```

非卖品，仅供非商业用途或交流学习使用

```
TERM. This Contract expires at the conclusion of code execution.
```

5.5.2 数据库合约

EOS 智能合约中的一个重要功能就是将某对象持久化存储，这就需要依托 multi_index 多索引表来实现，多索引表是一种在 RAM 中缓存状态或数据以便快速访问的方法。多索引表支持创建、读取、更新和删除（CRUD）操作，区块链不支持（它只支持创建和读取）。

多索引表提供快速访问数据存储，是存储数据以用于智能合约的实用方法。区块链记录事务，但应使用多索引表来存储应用程序数据。

它们是多索引表，因为它们支持在数据上使用多个索引，主索引类型必须是 uint64_t 并且必须是唯一的，但其他次要索引可以有重复。最多可以有 16 个附加索引，字段类型可以是 uint64_t、uint128_t、uint256_t、double 或 long double。

如果要对字符串建立索引，则需要将其转换为整数类型，并将结果存储在随后编制索引的字段中。下面我们结合具体案例教大家使用多索引表。

1. 创建一个结构

在使用多索引表之前，首先需要创建一个可以存储在多索引表中的数据结构，并定义索引获取函数（getter）用以获取该结构的索引值。请记住，结构中必须包含一个命名为“primary_key()”的 getter 函数，用以定义默认主索引，如果没有包含这个函数，编译器（eosiocpp）报“它找不到要用作主键的字段”错误。

如果你想拥有多个索引（最多允许 16 个索引），则为要索引的任何字段定义一个 getter，此时名称不太重要，因为你将 getter 名称传递给 typedef。

```
/// @abi table
    struct mystruct
    {
        uint64_t     key;
        uint64_t     secondid;
        std::string  name;
        std::string  account;

        uint64_t primary_key() const { return key; } // 定义默认主索引的getter函数
```

非卖品，仅供非商业用途或交流学习使用

```
        uint64_t by_id() const {return secondid; } // 定义额外索引的getter函数
    };
```

这里还有两点需要注意：

- 不要遗漏掉 ///@abi table，该部分用于告诉编译器 eosiocpp 你希望通过 abi 公开此表并使其在智能合约之外可见。
- 结构名称少于 12 个字符，并且全部为小写，否则编译器不能解析。

2. 利用 typedef 定义多索引表并定义索引

接下来需要定义将使用上面定义的指定结构的多索引表，并告诉它要索引的内容，以及如何获取正在索引的数据。该函数将自动创建一个默认主键，就是我们上面定义 primary_key()。因此如果我们想要一个只有主键的多索引表，请参考上文。

```
typedef eosio::multi_index<N(mystruct), mystruct> datastore;
```

这定义了一个名为 mystruct、存储的结构名称为“mystruct”的多索引表。N（mystruct）执行结构名到 uint64_t 的编译转换，这个 uint64_t 变量用于标识属于多索引表的数据。

若要添加其他索引或辅助索引，就需要使用 indexed_by 模板作为参数，用以定义该表的附加索引值。const_mem_fun 模板是一个适配器（adapter），该函数能将类的成员函数包装成仿函数使用，于是成员函数可以搭配各种泛型算法完成所谓的多态调用。

```
typedef eosio::multi_index<N(mystruct), mystruct, indexed_by<N(secondid),
    const_mem_fun<mystruct, uint64_t, &mystruct::by_id>>> datastore;
```

这里需要注意的一点是，结构名与表名必须匹配，而且 abi 文件中出现的名称遵循 12 个字符全部为小写的规则。

3. 智能合约示例

下面我们结合一个使用了多索引表的投票智能合约，通过这个示例介绍一下如何迭代查询表数据。

```
#include <eosiolib/eosio.hpp>

using namespace eosio;

class youvote : public contract {
//首先定义该类的成员变量
```

非卖品，仅供非商业用途或交流学习使用


```
    private:


        /// @abi table
//创建多索引表中存储的数据结构
        struct poll
        {
            uint64_t      key; // primary key
            uint64_t      pollId; // second key, non-unique, this table will
have dup rows for each poll because of option
            std::string   pollName; // name of poll
            uint8_t      pollStatus =0; // staus where 0 = closed, 1 = open,
2 = finished
            std::string  option; // the item you can vote for
             uint32_t     count =0; // the number of votes for each itme --
this to be pulled out to separte table.

            uint64_t primary_key() const { return key; }
            uint64_t by_pollId() const {return pollId; }
        };
//创建一个名为poll的多索引表，该表包括两个索引值：key与polld，其中key为主键，并且用
pollstable作为别名
            typedef  eosio::multi_index<N(poll), poll, indexed_by<N(pollId),
const_mem_fun<poll, uint64_t, &poll::by_pollId>>> pollstable;

        //声明名为_polls的polls table表
        pollstable _polls;


    public:
    //构造函数，初始化合约以及两张表
        youvote(account_name s):contract(s), _polls(s, s), _votes(s, s)
        {}

        /// @abi action
//定义版本查询函数，返回该智能合约版本并注册到abi文件中
        void version()
        {
            print("YouVote version  0.01");

        };

        /// @abi action
//
```

非卖品，仅供非商业用途或交流学习使用


```
        void addpoll(account_name s, std::string pollName)
        {
            //require_auth(s);这里可以设置一个权限判断，只有账户名为s的用户可以调用该
              action

            print("Add poll ", pollName);

            //在表中创建一个新对象，其中支付存储空间花费者为当前调用该action的用户，注
              入对象的pollName为参数中输入的poll  Name，pollStatus、option、count
              均设为空
            _polls.emplace(get_self(), [&](auto& p)
                                        {
                                    p.key = _polls.available_primary_key();
                                    p.pollId = _polls.available_primary_key();
                                    p.pollName = pollName;
                                    p.pollStatus = 0;
                                    p.option = "";
                                    p.count = 0;
                                });
        };
        /// @abi action
//定义移除函数，删除指定账户名在表中的所有对象
        void rmpoll(account_name s, std::string pollName)
        {
            //require_auth(s);

            print("Remove poll ", pollName);
//声明一个vector，用以存储检索后的返回值
            std::vector<uint64_t> keysForDeletion;
// 通过for循环，利用item迭代器逐个调用_polls表中的每个元素
            for(auto& item : _polls)
            {
//若当前迭代器指代的对象pollName等于输入参数中的pollName，则将迭代器键值压入vector中
                if (item.pollName == pollName)
                {
                    keysForDeletion.push_back(item.key);
                }
            }

// 逐个调用vector中保存的键值对应的表中数据
            for (uint64_t key : keysForDeletion)
            {
                print("remove from _polls ", key);
//查找key键值的对象，并返回索引
```

非卖品，仅供非商业用途或交流学习使用


```
auto itr = _polls.find(key);
                if (itr != _polls.end())
                {
//删除该索引对应对象
                _polls.erase(itr);
            }
        }
};
//在ABI文件中注册action
EOSIO_ABI( youvote, (version)(addpoll)(rmpoll))
```

4. 智能合约测试

接下来我们测试一下合约，看看是否按我们的期望准确执行，不做重复说明，我们已经将上述合约部署上链，合约账户名为 testyouvote，我们下面测试一下其基本功能。

首先调用一下 version 函数。

```
cleos push action testyouvote version '[]' -p  testyouvote@active
```

我们看到返回值为 0.01，返回正确。

```
executed transaction: c3dc74e42c551982c7778c077e02a614a2eca4593624bd4d2706d9
    cabc2fb0f9  96 bytes  15037 us
#   testyouvote <= testyouvote::version         ""
>> YouVote version  0.01
```

接下来我们测试一下 addpoll 函数。

```
cleos push action testyouvote addpoll '["eosio","test1"]' -p  eosio@active
```

执行结果如下，我们可以看出成功打印出 Add poll test1 信息，添加信息成功。

```
executed transaction: ecf3dd71480ca367bc60ed2b6fe3f3250f4196b0386bf54034c768
    111626d607  112 bytes  1023 us
#   testyouvote <= testyouvote::addpoll     {"s":"eosio","pollName":"test1"}
>> Add poll test1
```

最后我们测试一下 rmpoll 函数。

```
cleos push action testyouvote rmpoll '["eosio","test1"]' -p  eosio@active
```

执行结果如下，我们可以看出成功打印出 Remove poll test1 remove from _polls 0 信息，删除信息成功。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
executed transaction: e56a40a0edba0126425567de9e6f13d52ea4b4c4a5aba09f60762b
    9255b55f5c  112 bytes  1042 us
#   testyouvote <= testyouvote::rmpoll        {"s":"eosio","pollName":"test1"}
>> Remove poll test1remove from _polls 0
```

5.6　常见 bug 与 debug 思路总结

下面我们介绍一下常见的 bug，以及设计 EOS 智能合约必须要注意的地方。

1. 整型溢出错误

这已经不是一个新 bug 了，我们知道在计算机中，整数分为无符号整数和有符号整数两种。其中有符号整数会在最高位用 0 表示正数，用 1 表示负数，而无符号整数则没有这种限制。另外，我们常见的整数类型有 8 位（单字节字符、布尔类型）、16 位（短整型）、32 位（长整型）等。关于整数溢出，其实它与其他类型的溢出一样，都是将数据放入了比它本身小的存储空间中，从而出现了溢出。由此引发的一切程序漏洞都可以成为整数溢出漏洞。所以设计具有转账功能的智能合约并使用自己的数据结构描述代币，就必须在对代币数值进行算数运算时进行安全检查。否则当出现误操作时容易产生整型溢出错误，可能导致代币量归零甚至变成负数的严重后果。

解决方案：在涉及新的代币类型创建以及新的转账相关函数的合约开发时，需要实现对数值类型进行安全检查，我们参照一下 eosio.Token 合约中的资产安全检查流程。首先验证资产标识合法性，然后验证资产数量的变量类型的合法性，最后判断总量是否为正值。

所需函数：

```
eosio_assert( sym.is_valid(), "invalid symbol name" );
eosio_assert( maximum_supply.is_valid(), "invalid supply");
eosio_assert( maximum_supply.amount > 0, "max-supply must be PoSitive");
```

2. 权限检查不严谨

权限检查不严谨，造成逻辑漏洞。有很多时候，智能合约中 action 需要设置权限，只有具备调用权限的账户才可以发送 action 调用该函数，所以权限设置的缺失或者考虑不够严谨很可能会造成所有用户都具有调用权限，这也是十分危险的。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

解决方案：在设计函数的时候需要仔细考虑该函数的调用权限，以避免攻击者的恶意调用。

所需函数：

```
require_auth( _self );
has_auth( _self );
```

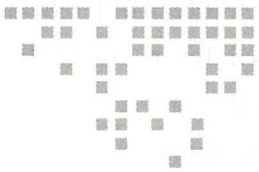
3. 常见代码错误

这里指的是开发者要注意 EOS API 函数的参数类型。第一个参数传入的整型变量需要小于 256，若使用该 API 前未对输入进行检查，则可能导致整型溢出，从而导致操作了错误类型的代币，带来严重后果。例如数据库 API 使用不严谨，如 multi_index 中提供的 get 和 find。其中 get 会检查数据是否查询成功，数据未找到则断言退出，而 find 不会检查数据查询情况，需要用户自行判断，如果缺少判断直接使用将会导致指针使用问题。

解决方案：在调用 API 之前应仔细阅读 API 文档，做好入参类型匹配。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用



第 6 章 Chapter 6

基于 EOS 的 DApp 开发

本章将详细介绍基于区块链的 DApp 的设计思想与开发流程，旨在帮助读者了解 DApp 的概念以及设计思维。

6.1 什么是 DApp

DApp 是去中心化应用程序的缩写形式。一般对于 DApp 的定义是，运行在分布式网络上，参与者的信息被安全保护（也可能是匿名的），通过网络节点进行去中心化操作的应用。应当注意的是，去中心化应用程序并不等于智能合约，主要在三个关键方面有所不同：

- 去中心化应用程序应该适用于所有应用场景；
- 去中心化应用程序不局限在金融方面；
- 去中心化应用程序需要有前端代码和用任何语言编写的用户界面（就像应用程序一样），可以调用作为程序后端的智能合约。此外，它的前端最好的选择是托管在去中心化存储网络中，例如 Swarm 或 IPFS。

所以总结下来就是一句话：DApp = 前端 + 智能合约（+ 其他支持类设备），对于被视为 DApp 的应用程序，它必须具有以下几点特征：

仅供非商业用途或交流学习使用

- 去中心化应用程序必须是完全开源的，它必须自主运行，并且没有实体控制其大部分 Token。应用程序可以根据提议的改进和市场反馈调整其协议，但所有更改必须通过其用户的共识来决定。
- 去中心化应用程序的数据和操作记录必须以加密方式存储在公有区块链中，以避免任何中心化问题。
- 去中心化应用程序必须使用 Token（以太坊等公链代币或其系统本身自发行的 Token），这对于用户使用应用程序来说是必需的，因为持币证明了用户真正为 DApp 的发展做了贡献，相当于贡献量证明，并且应用程序应该结合 Token 设置激励机制，以奖励用户任何有价值的贡献。
- 去中心化应用程序必须根据标准的密码算法生成 Token，该算法充当节点对应用程序有贡献的证据（比特币使用工作证明算法）。

6.1.1 比特币是第一个 DApp

比特币的创造者中本聪（Satoshi Nakamoto）将他的发明描述为“对等电子现金系统”。通过使用点对点的分布式账本技术，比特币已被证明可以有效地解决无信任和可扩展电子现金系统带来的问题。然而，除了作为点对点电子现金系统之外，比特币也是用户可以通过计算机软件进行交互的应用程序。基于这样的标准，我们可以说比特币就是一种去中心化应用程序。原因如下：

- 所有比特币软件应用程序都是开源的，没有任何实体（政府、公司或组织）控制比特币，所有与比特币使用相关的记录都是开放和公开的。
- 比特币使用无法更改的预定算法生成其 Token，这些 Token 是比特币运行所必需的。比特币矿工因其在保护比特币网络方面的贡献而获得比特币奖励。
- 比特币的所有变更必须通过工作量证明机制获得其用户的多数共识。

6.1.2 DApp 命名的原因

在 EOS 创始人 Daniel Larimer 撰写的一篇文章中，去中心化应用程序最初被描述为分布式自治公司（DAC）。但是我们尽量避免使用“公司”一词有如下两个原因。

首先，因为它带有不必要的歧义。例如，一个合法公司需要拥有股份、一个 CEO、员工等。像比特币一样的 DApp 显然没有这些特征。此外，考虑到命名对于各

仅供非商业用途或交流学习使用

种国家对 DApp 的看法非常重要。我们知道由于比特币在某种程度上具有了货币的概念，政府努力思考该如何管理比特币。如果使用“公司”这个名词，政府可能不得不继续去考虑管理作为去中心化应用的开源计算机程序，为技术的发展设置一些不必要的枷锁。

其次，因为传统公司可能会采用多种技术来筹集资金（例如出售其股票并支付股息或借入股票并支付利息），然而 DApp 不需要。DApp 的概念是如此强大和优雅，因为它不包括这些传统的企业模式。对于一个 DApp 来说，其 Token 的所有权是确认用户是否可以使用该去中心化应用所需的全部信息。就这么简单。Token 的价值取决于人们对应用程序的重视程度。所有激励措施、所有货币化、所有提升支持的方式都建立在这个美丽而又简单的结构中。DApp 不需要重新创建集中式公司过去必需的功能，以平衡股东的权力并为投资者和员工提供回报。

6.1.3　DApp 运作的法律模型

在开源许可下运营使得 DApp 可以在没有版权或专利限制的情况下开放创新。此外，通过完全开源，去中心化应用程序可以在开源软件的法律模型下运行。例如，比特币使用 MIT 开源软件许可证。类似地，主协议要求所有基于它的代码都是开源的并且可供社区使用。

1. 发行和持有代币

从技术角度来看，作为群体销售的一部分发行 Token 的人正在销售对该去中心化应用程序的使用权。与用户购买的 Token 相关联的私钥实际上是用户访问 DApp 软件所需的密码。从税收的角度来看，持有代币的人持有数字财产。如果 Token 在 DApp 中使用之外没有市场价值，则很难确定它们的实际价值。

由于很少有司法管辖区公开就如何从监管和税务角度处理 DApp 发行的代币提供指导，因此应咨询特定司法管辖区的法律专家。

2. 非营利机构

DApp 没有法律实体可以运营，因为它不是公司。代币的所有者不需要由公司代表，贡献者也不需要任何特定的法律实体。但是，有时 Token 是由一个永远不会从 DApp 获得经济利益的非营利组织发行的。这样的组织将承担以下责任：

仅供非商业用途或交流学习使用

- ❑ 发行初始代币。
- ❑ 持有开发人员 Token。
- ❑ 管理赏金付款。
- ❑ 确定 DApp 方向。

理想情况下，非营利组织将以分散的方式做出决策，使用“股权证明”投票机制进行任何决策。

DApp 的分类是灵活的，每个人都有自己的一套分类标准，比如根据 DApp 应用场景等因素分类。但在这里，考虑到分类结果的全面性，我们可以根据 DApp 是否拥有自己的区块链或使用另一个 DApp 来对 DApp 进行分类。基于此标准，有三种类型的 DApp。

类型 1：去中心化应用程序有自己的底层区块链，或者说它自己就是一个底层公链。比特币是类型 1 的去中心化应用程序中最著名的例子，与之相似还有以太坊、EOS 等。

类型 2：去中心化应用程序必须基于某种已经存在的区块链来运行，且类型 2 去中心化应用程序一般是协议，它定义了一种某功能的可行且安全的标准。类型 2 的 DApp 一般面向的目标用户是开发者，ERC20、ERC721 协议是类型 2 去中心化应用程序的一个例子，它们用以安全的发行不同种类的数字加密货币。

类型 3：去中心化应用程序使用类型 2 去中心化应用程序的协议，且需要有人类易用的友好界面或前端，这类 DApp 主要面向的目标用户是消费者端，提供的服务也相对具体且垂直，主要解决消费者的真实需求。类型 3 又可以根据应用场景细分为游戏、资产交易、数字收藏品、博彩以及一些其他方向。

6.2 开发 DApp 的一般流程

本节将详细介绍 DApp 开发的一般流程，手把手带读者了解 DApp 开发的通用流程。

6.2.1 建立共识的机制

首先你的开发团队需要设计好待开发 DApp 的共识机制，看到这里，你可能会有些茫然，区块链不是有共识机制吗？那为什么基于区块链开发的 DApp 还需要设

仅供非商业用途或交流学习使用

计共识机制呢（其实区块链本身就是一种去中心化应用）？答案很简单，任何去中心化应用程序都不应该属于某个独立第三方，而是应该属于一个类似于董事会的社群。DApp 的每一次更新升级，或者说其发展道路是应该由社群来决定的而非一个 CEO 来决定，那么，在这个复杂的社群中如何对某个决策达成一致决定呢？这就需要共识机制了。

下面我们介绍一下常见的 DApp 共识机制，必须要说的是，现在大部分 DApp 采用混合式共识机制，但是在这里我们分别介绍每一种共识机制的概念。

- 工作量证明 PoW：我们知道比特币、以太坊等公链采用的共识算法就是 PoW，即在这个系统中，为网络安全做的工作越多，按理来说，回报就应该更高，多劳多得，公平合理。采用这一共识算法的 DApp 一般会为社群指定工作量考量规则，为 DApp 发展做了更多工作的用户将会在决策或分红中获得更大的权重。
- 贡献量证明 PoC：贡献量证明这个概念与工作量证明机制类似，即为 DApp 贡献更多的用户将会在决策或分红中获得更大权重，采用这种共识算法的 DApp 需要将“贡献”定义清楚，就像 steemit 一样，用户的贡献就是在平台上发布了多少优质评论，并根据评论获得的点赞数量量化贡献大小，并返回相应数量的 Token。这种模式灵活度更高，而且可以培养开发团队与社群之间的黏性，使双方成为利益共同体。
- 权益证明 PoS：权益证明与前两种方式相比就显得粗暴的多，持有自身 Token 数量越多，拥有的权重越高。因为如果用户花更多的钱用来持有自身 Token，那么从经济激励的角度考虑，该用户就越没有作恶的动力，就会做出有益于 DApp 发展的正确决策。

6.2.2　设计分发 Token 的机制

Token 这个概念可能是当前区块链中最重要的概念了，现在通常翻译成“通证”，顾名思义，就是可流通的权益证明。元道先生的解释最为透彻到点：Token 是区块链上可流通的加密数字权益证明，Token 的目的是允许持有者访问指定的去中心化应用程序提供的服务。例如，个人必须拥有多个比特币才能在比特币网络上执行任何交易。DApp 中的代币不代表任何相关资产，它们不赋予股息权利，也不代表股权。虽然

仅供非商业用途或交流学习使用

DAppToken 的价值可能会随着时间的推移而增加或减少，但 Token 不是股本证券。可以说一个 DApp 项目是否成功的关键评价标准就是其 Token 的市场价值。

所以 DApp 项目中最为关键的流程就是 Token 的分发，DApp 项目方在开发初期就需要仔细考虑其 Token 的设计，包括 Token 总量模型以及 Token 的分发机制。

其中 Token 总量模型决定了 Token 的市场价格，来源于一个最简单的金融学知识——价值等于总价值除以总量，因此必须仔细考虑 Token 的总量模型，常见的有两种总量模型。

1. Token 总量有上限

应用型 Token 的设计普遍采取积分 + 股份相结合的形式，可以完全把 Token 看作项目所有权的代表。应用型 Token 总量固定，一方面，类似积分的特性，许多团队许诺 Token 持有者可以享受使用 Token 换取某种服务或抵扣某种费用的权利，比如某些交易所提出的 Token 抵扣交易手续费等；此外，团队还可以制定运营方案，利用积分发放的方式来免费发放 Token，以此提升社区活跃度。另一方面，因为 Token 具备股份特性，使得 Token 的持有者可以通过项目团队主导的 Token 分红、Token 回购销毁等方式来享有项目发展的红利。在为应用设计 Token 时，要注意应用的市值通常较小，大部分市值可能落在 1 亿～100 亿，因此，应用型 Token 的发行数量不宜超过 100 亿，应根据估值将总币量设定在 1000 万～100 亿，以便以法币计价时，每个币的价格在 0.01-1000 元之间，有利于投资者间的沟通和交易。

以 BNB（币安币）为例：BNB 总量被限定为 2 亿个，50% 归 ICO 投资人所有，40% 归团队，剩下 10% 归私募投资人所有。持有 BNB 的用户，在币安平台上交易，在前 5 年可以享有逐年递减的手续费折扣优惠。此外，项目团队会在每季度拿出当季净利润的 20% 用于回购 BNB，并将回购来的 BNB 销毁，直至所有 BNB 总量为 1 亿个为止。

设计应用总量有上限的 Token 时一共有 4 个可调整的关键参数。

- ❑ Token 总量。
- ❑ 投资人、团队、基金会、矿工等利益相关群体的分配比例。
- ❑ 积分运营方案：运营活动中 Token 的免费发放 / 使用系统时实行费用抵扣。
- ❑ 股份分红方案：将净利润用于回购 Token 再销毁 / 将净利润按照持币比例直接分配给持币人（净利润通常为另一种 Token，比如 BTC、ETH 等）。

仅供非商业用途或交流学习使用

2. Token 总量无上限（股份增发）

无论是现实中的公司，还是区块链项目，都可能存在再融资的需求，这时可能面临配股、增发或是发行可转债等情景。由于应用型 Token 基于公链创建，所有特性受制于公链所能提供的功能。比如在公链项目 Waves 中，提供了资产增发的选项，基于 Waves 发行的应用型 Token，均可实现 Token 增发的功能。

另一方面，Token 的分发机制也需要做详细考虑。因为确定总量后，你总得把这些 Token 分发出来。一个优秀的 Token 分发机制可以保证项目的可持续发展，可以通过三种常见机制分配其代币：挖矿、募资和开发，而且在设计 Token 释放计划时需要充分考虑这三种分配机制。

- 挖掘机制：Token 分发给那些为 DApp 操作贡献最多工作的人。以比特币为例，比特币通过预定算法分发给矿工，验证交易并维护比特币区块链。
- 募资机制：将代币分发给那些资助 DApp 初期发展的人。这里涉及一个很重要的概念——私募，就是一个区块链项目在进行 ICO 之前进行的非公开融资，私募者资助的数字货币（常用 ETH）将支持项目早期开发过程的开销，项目方将承诺在项目正式开发完成后，按照一个高于市场价格的返还比例返回资助者指定数量的项目 Token。
- 奖励机制：可以设置一定的奖励机制激励贡献者继续提供贡献，这是 DApp 是否可以充分利用社群力量达成自身可持续发展的关键因素。举个例子，由于 DApp 项目需要开源，一个优秀开源项目一定是可以吸引大量开发者共同开发的，所以为了奖励这部分开发者，可以设置相应的奖励机制用于奖励这些代码贡献者。

6.2.3　常见 DApp 开发阶段

去中心化应用程序的开发分三个步骤进行。

1. 发布描述 DApp 及其功能的白皮书

与比特币的情况一样，DApp 最常见的形式是公开发布描述协议、其功能和实现的白皮书。公开发布后，社区的反馈对于发展议程的进一步发展是必要的。

2. 分发初始 Token

如果 DApp 使用挖掘机制来分发其 Token，则会发布参考软件程序，以便可以将

其用于挖掘。在比特币的情况下，发布了参考软件程序并创建了初始事务块。如果DApp正在使用筹资机制，则DApp的利益相关者可以使用钱包软件，以便他们可以交换DApp的Token。如果DApp正在使用奖励机制，则会建立一个赏金系统，允许执行任务的建议，跟踪正在执行这些任务的人员以及奖励的标准。

3. DApp的所有权权益被分散

随着采矿、筹款和奖励的代币分发给更多的参与者，DApp的所有权变得越来越分散，而在早期持有多数股权的参与者控制越来越少。随着DApp的成熟，具有更多不同技能的参与者将被激励做出有价值的贡献，并且DApp的所有权将进一步分配。通过市场力量，DApp的代币被转移给那些最重视它的人。那些人可以在他们拥有专业知识的领域为DApp的发展做出贡献。

比特币的情况说明了这一点。根据一些估计，Satoshi Nakamoto开采了前1 000 000比特币中的许多比特币。随着开发人员为比特币贡献代码并且矿工为比特币网络贡献了计算能力，市场开始更加重视比特币。随着系统日趋成熟，具有不同技能的人们开始重视比特币并为其发展做出贡献。现在超过1200万比特币正在流通，而中本聪的原始所有权股份已被稀释。

6.3 DApp的应用领域

下面我们结合几个优秀项目，介绍一下当前DApp的几大主流应用领域。

6.3.1 游戏领域

区块链游戏一直以来都是DApp得到重用的应用场景，2017年一款名为《加密猫》CryptoKitties的以太坊游戏突然风靡全球，玩家数量和交易暴增直接导致以太坊几近瘫痪。

首先我们来思考一下为什么DApp第一个落地的领域是游戏呢？原因有三。

- 传统游戏领域本身就存在自身的代币模型，以腾讯系游戏为例，玩家可以使用Q币在任意一个腾讯游戏中购买装备、角色、皮肤等，且这些购买品已经具有了数字资产的属性，这种模式已经被大部分玩家所接受，且更容易实现Token化，所以相较于其他领域，以游戏作为DApp的切入口不会遇到太大的阻力，玩家相对来说更容易接受。

- 传统游戏领域中大部分游戏的数字资产均不可流通，交易通常只能发生在系统与玩家之间，这在一定程度上降低了游戏的可玩性。
- 游戏领域相较于金融领域不太敏感，易于推广。

下面我们将对区块链游戏做一个详细的区分，希望能够加深你对区块链游戏的理解。

数字收藏品游戏是区块链游戏最典型的例子，以日活量最大的 CryptoKitties 为例。在 CryptoKitties 中，它的玩法很简单，类似于其他可收集的纸牌游戏（如口袋妖怪）。玩家可以购买、出售、交易和培育他们自己可爱的虚拟猫，每一只虚拟猫都是以太坊网络中独一无二的数据集，而区块链又使这个数据集具有了不可篡改、永久留存的特性，所以这不仅仅构建了一种全新且有趣的收藏方式，也让非程序员与金融从业者第一次能够直观的享受区块链带来的服务。

如图 6-1 所示，Cryptokitties 目前已经获得了超过 200 万美元的加密货币。与此相似的同类数字收藏品游戏大体上不外乎增加一些新鲜的玩法（如对战等），游戏整体类似于早期页游模式（如电子宠物等）。所以可以看得出，这种游戏模式已经被越来越多的用户认可和喜欢。

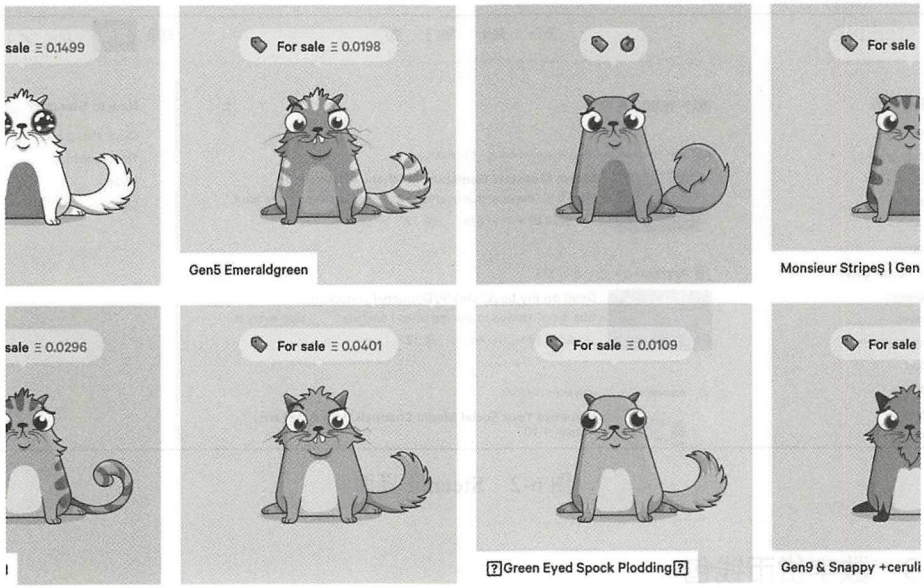


图 6-1 《加密猫》游戏界面

6.3.2 社交 / 内容分发平台

社交媒体是现代人类沟通的支柱，很多人每天花在社交平台上交流的时间甚至超过真实生活中的交流时间，大量的内容分发平台基本上也是所有人主要的信息来源，所以这一领域庞大的用户量与日活对于很多区块链开发者来说具有巨大吸引力，且区块链技术确实有解决传统中心化平台痛点的潜力。那么，中心化的社交与内容分发平台都有哪些痛点可以被解决呢？

通常来说主要有两点，一是用户数据隐私保护力度不够，二是平台用户收益较低，流量产生的主要利润被平台赚取，用户除了可以享受快捷通信带来的服务之外并无任何收益。区块链在验证用户身份过程中所展示出来的匿名性优势，以及通证经济带来的多方共赢优势对于解决以上痛点来说再合适不过了！

其中一个具有代表性的案例就是 Steemit，如图 6-2 所示。Steemit 是一个用区块链技术搭建的社交网络。这个网络会用 Steemit 的代币来奖励内容生产者，说白了就是为优质内容付费！ Steemit 的目的是让这个注意力经济中的用户人群也可以受益。以前我们在各大社交媒体上面的注意力都被运营商收集起来卖给广告商了，现在我们可以把自己的注意力用来奖赏别人或者制造财富。

图 6-2 Steemit 界面

6.3.3 数字货币钱包

如果你有数字货币，你就需要记住你的私钥值，如果私钥丢失那就意味着这笔资

非卖品，仅供非商业用途或交流学习使用

产永远丢失，这样就需要一个软件或者硬件来管理你的数字货币，它被称为数字货币钱包。通常情况下数字货币钱包应该具有如下功能：

- 保存并管理用户公私钥；
- 查询指定地址区块链上数据，如余额等；
- 具有转账功能，内置加密算法，可以对交易进行签名。

如图 6-3 所示，Coinbase 是当前最受欢迎的钱包之一。Coinbase 是一款一体化数字货币工具，用户可以在这个平台上购买、使用和存储加密货币。Coinbase 可作为网络应用程序使用，也可在 IOS 和 Play 商店中使用，移动版本也有很好的评价。

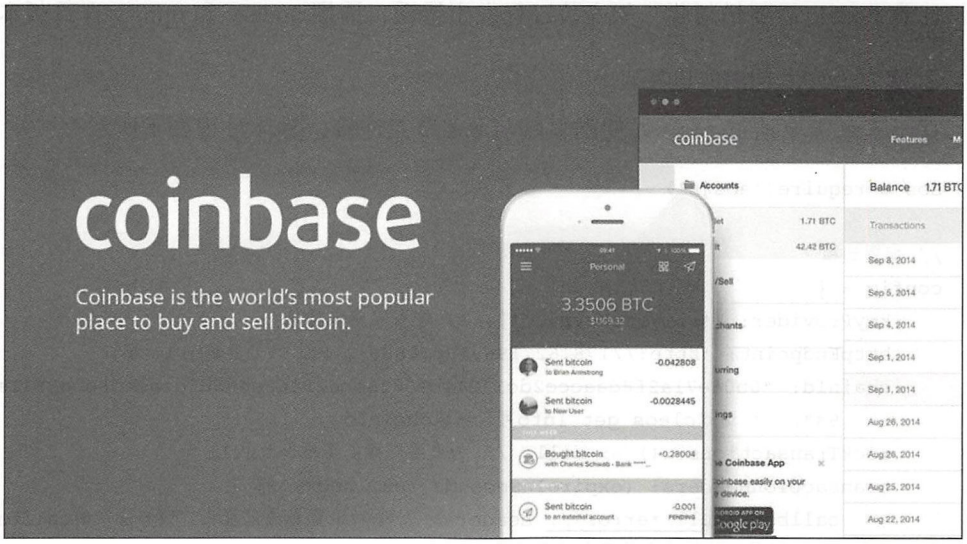


图 6-3　Coinbase 钱包界面

以上三种应用场景是目前 DApp 落地数量最多的三个场景，当然还有很多优秀的构想，如供应链管理、医疗数据管理、计算资源共享等，只不过受当前区块链技术现状、用户接受度以及相关政策等限制，并没有比较成功的案例，但是值得我们跟进与思考，这里不做展开。

6.4　基于 EOS 的 DApp 开发思路

下面我们结合具体案例，详细叙述一下如何开发一个基于区块链的 DApp。经过

非卖品，仅供非商业用途或交流学习使用

上文的介绍我们知道，一个简单的 DApp 由两部分组成：前端界面 / 客户端 + 后端智能合约，两者之间的信息交互就依赖于 eosjs 库，eosjs 提供了使用 js 代码调用 eosio 相关 api 接口，开发者可以利用这个库，实现前后端相连，下面我们介绍一些该库提供的常用方法。

1. 安装 eosjs

在 nodejs 中安装 eosjs 的流程如下：

```
npm install eosjs
npm install eosjs-api
```

注意 nodejs 需要最新版，如安装过程发生错误，请用 node-v 和 npm-v 查看版本。

2. 建立 eosjs 与链的连接

接下来就需要通过 eosjs 库使网页端与 eos 节点连接，具体过程如下所示。

```
Eos = require('eosjs')

// 配置选项
config = {
    keyProvider: ['PrivateKeys...'], // 配置私钥字符串
    httpEndpoint: 'http://178.62.196.196:8888', //DEV开发链url与端口
    chainId: "0b08e71a2f8caaccc2dc13244b788f5eba29462ecd5d5dea1ad8cbe9581e88
        5a", // 通过cleos get info可以获取chainId
    mockTransactions: () => null, // 如果要广播，需要设为null
    transactionHeaders: (expireInSeconds, callback) => {
        callback(null/*error*/, headers) //手动设置交易记录头，该方法中的callback
            回调函数每次交易都会被调用
    },
    expireInSeconds: 60,
    broadcast: true,
    debug: false,
    sign: true,
    authorization: null // 该参数用于在多签名情况下，识别签名账号与权限，格式如：
        account@permission
}

eos = Eos(config)
```

3. 建立 eosjs-api 与链的连接

如果加载了 eosjs 后，默认加载 eosjs-api，无需单独链接 eosjs-api 与链。

非卖品，仅供非商业用途或交流学习使用

```
EosApi = require('eosjs-api')

// everything is optional
options = {
    httpEndpoint: 'http://178.62.196.196:8888',
    verbose: false, // API logging
    logger: { // Default logging functions
        //log: config.verbose   console.log : '',
        error: console.error
    },
    fetchConfiguration: {}
}

eos = EosApi(options)
```

4. 获取帮助

不添加如何参数，可以获取该方法的使用说明。

```
eos.getBlock()
```

5. 获取链上最新出块信息

通过如下代码获取链上最新出块信息。

```
eos.getInfo({}).then(result => {
    console.log(result)
})
```

返回：

```
{ server_version: 'b195012b',
    chain_id: '0b08e71a2f8caaccc2dc13244b788f5eba29462ecd5d5dea1ad8cbe9581e885a',
    head_block_num: 209307,
    last_irreversible_block_num: 209267,
    last_irreversible_block_id: '00033173a9ccd4bdd60a92d257e9354023b0457b134
        797be472a236cd908bc31',
    head_block_id: '0003319ba8ddc60d80c3cd0c7a70695cfd951f51ace9a798c913384c
        fbae659c',
    head_block_time: '2018-07-15T01:51:07.500',
    head_block_producer: 'eoshackathon',
    virtual_block_cpu_limit: 100000000,
    virtual_block_net_limit: 1048576000,
    block_cpu_limit: 99900,
    block_net_limit: 1048576
}
```

非卖品，仅供非商业用途或交流学习使用

如果需要拿到某一个数据值，比如 head_block_producer，则使用：

```
eos.getInfo({}).then(result => {
    console.log(result.head_block_producer) //以对象属性方式获取head_block_producer
})
```

6. 获取指定区块信息

获取指定区块信息方法如下。

```
eos.getBlock(200000).then(result => { console.log(result) }) //获取第200000个区块
```

或者

```
eos.getBlock({block_num_or_id: 200000}).then(result => { console.log(result)
    }) //获取第200000个区块
```

或者

```
eos.getBlock('00030d4011a6744857533a6e6d907037a94c27a2dc006b4d28125f76bed
    2b355').then(result => { console.log(result) }) //根据id获取区块
```

或者

```
eos.getBlock({block_num_or_id: '00030d4011a6744857533a6e6d907037a94c27a2dc00
    6b4d28125f76bed2b355'}).then(result => { console.log(result) }) //根据id
    获取区块
```

或者将返回值传到回调函数 callback 中处理。

```
callback = (err, res) => { err ? console.error(err) : console.log(res) }
eos.getBlock(200000, callback)
```

7. 获取账户余额

获取账户余额代码如下：

```
eos.getCurrencyBalance({ code: "eosio.Token", account: "eosio", symbol:
    "DEV" }).then(result => console.log(result))
```

以上命令相当于：

```
cleos get currency balance eosio.Token eosio DEV
```

8. 获取某代币的信息

获取某代币信息的代码如下：

```
eos.getCurrencyStats({code: "eosio.Token", symbol: "DEV"}, callback) //用上面
    出现的callback
```

非卖品，仅供非商业用途或交流学习使用

9. 获取账户信息

获取某指定账户信息代码如下：

```
eos.getAccount({account_name: "eoshackathon"}, callback) //获取eoshackathon
    账户的信息
```

相当于

```
cleos get account eoshackathon
```

返回

```
{ account_name: 'eoshackathon',
  head_block_num: 219997,
  head_block_time: '2018-07-15T03:20:12.500',
  privileged: false,
  last_code_update: '1970-01-01T00:00:00.000',
  created: '2018-07-13T20:54:28.000',
  ram_quota: 8148,
  net_weight: 500000,
  cpu_weight: 500000,
  net_limit: { used: 145, available: 120795486, max: 120795631 },
  cpu_limit: { used: 1511, available: 11518458, max: 11519969 },
  ram_usage: 3414,
  permissions:
    [ { perm_name: 'active', parent: 'owner', required_auth: [Object] },
      { perm_name: 'owner', parent: '', required_auth: [Object] } ],
  total_resources:
      { owner: 'eoshackathon',
        net_weight: '50.0000 DEV',
        cpu_weight: '50.0000 DEV',
        ram_bytes: 8148 },
  self_delegated_bandwidth: null,
  refund_request: null,
  voter_info: null
}
```

10. 获取智能合约代码

获取 eosio 账户的所有合约代码如下：

```
eos.getCode({ account_name: "eosio"}, callback)
```

11. 获取智能合约 ABI

获取智能合约 ABI 的代码如下：

非卖品，仅供非商业用途或交流学习使用


```
eos.getAbi({ account_name: "eosio"}, callback)
```

12. 获取 Table 行数据

获取 table 中行数据的代码如下：

```
eosjs.getTableRows({"scope":contract_name, "code":contract_name, "table":
    "game", "json": true}).then(result => {}
```

13. 获取账户的 Actions 列表

获取某账户 Actions 列表的代码如下：

```
eos.getActions({account_name: "eoshackathon", PoS: 0, offset: 15}, callback)
    //PoS和offset是指：从第PoS条记录开始获取offset条Actions
```

14. 获取公钥对应的账户

获取公钥对应账户的代码如下：

```
eos.getKeyAccounts({ public_key: 公钥字符串}, callback)
```

如果查找到账号，则返回 []，如果该公钥没有对应账号，则报错。

```
cleos get accounts 公钥
```

15. 获取主账号控制的其他账号

获取主账户控制的其他账号信息的代码如下：

```
eos.getControlledAccounts({ controlling_account: "eoshackathon"}, callback)
```

16. 获取 transaction 交易细节

该指令有 bug，慎用！

```
eos.getTransaction({id: "xxxx"}, callback)
```

17. 转账交易

首先，在链接配置 config 中，keyProvider 表示发送方私钥；其次，可以按如下方式设置 options 参数。

```
options = {
    authorization: '发送方账号@active',
    broadcast: true,
    sign: true
}
```

完成转账交易的方式如下。

非卖品，仅供非商业用途或交流学习使用

方式 1：通过 transfer 函数完成转账。

```
eos.transfer('发送方账号', '接收方账号', '0.3000 DEV','memo')
```

如果需要对返回值处理：

```
eos.transfer('发送方账号', '接收方账号', '0.3000 DEV','memo', callback)
```

如果有 options 参数，则：

```
eos.transfer('发送方账号', '接收方账号', '0.3000 DEV','memo', options, callback)
```

方式 2：使用对象完成转账。

```
eos.transfer({ from: '发送方账号', to: '接收方账号', quantity: '0.1000 DEV',
    memo: '备注', callback })
```

如果不想广播交易，可以使用以下简便指令：

```
eos.transfer('发送方账号', '接收方账号', '0.3000 DEV','memo', false) //在最后加上
    false，不广播
```

方式 3：使用 eos.transaction，构造对象执行如下命令。

```
eos.transaction(
    {
        // ...headers,
        actions: [
            {
                account: 'eosio.Token',
                name: 'transfer',
                authorization: [{
                    actor: '发送方账号',
                    permission: 'active'
                }],
                data: {
                    from: '发送方账号',
                    to: '接收方账号',
                    quantity: '0.3000 DEV',
                    memo: '备注'
                }
            }
        ]
    }
    // options -- example: {broadcast: false}
)
```

以上命令与以下 cleos 相同：

非卖品，仅供非商业用途或交流学习使用


```
cleos push action eosio.Token transfer '[ "发送方账号", "接收方账号",  "0.3000
    DEV", "备注" ]' -p 发送方账号
```

18. 新建账号

通过 eosjs 创建新账号的代码如下：

```
creatoraccount = "testtesttest" //主账号
newaccount = "testtest1113" //新账号
newaccount_pubkey = "EOS5LUYdLZAd3uHF5XGAeE61aTeSXWqvqxBSUq3uhqYH7kY15Drjf"
    //新账号的公钥

//构建transaction对象
eos.transaction(tr => {
    //新建账号
    tr.newaccount({
        creator: creatoraccount,
        name: newaccount,
        owner: newaccount_pubkey,
        active: newaccount_pubkey
    })

    //为新账号充值RAM
    tr.buyrambytes({
        payer: creatoraccount,
        receiver: newaccount,
        bytes: 8192
    })

    //为新账号抵押CPU和NET资源
    tr.delegatebw({
        from: creatoraccount,
        receiver: newaccount,
        stake_net_quantity: '1.0000 DEV',
        stake_cpu_quantity: '1.0000 DEV',
        transfer: 0
    })
})
```

19. 购买 RAM

通过 eosjs 为指定账户购买 RAM 的代码如下：

```
creatoraccount = "testtesttest" //主账号
newaccount = "testtest1113" //新账号

eos.transaction(tr => {
```

非卖品，仅供非商业用途或交流学习使用


```
    tr.buyrambytes({
        payer: creatoraccount,
        receiver: newaccount,
        bytes: 8192
    })
```

或者

```
eos.transaction(tr => {
    tr.buyram({
        payer: creatoraccount,
        receiver: newaccount,
        quant: 8 //以k为单位的内存，8k=8192字节
    })
```

20. 出售 RAM

通过 eosjs 实现出售 RAM 功能的代码如下：

```
eos.transaction(tr => {
    tr.sellram({
        account: 'testtesttest',
        bytes: 1024 //出售1k内存
    })
})
```

21. 竞拍账号名

通过 eosjs 竞拍指定账号名的代码如下：

```
eos.transaction(tr => {
    tr.bidname ({
        bidder: "testtesttest",
        newname: "竞拍的名字",
        bid: 价格
    })
})
```

22. 抵押 CPU 和 Net

通过 eosjs 为指定账户抵押 CPU 与 Net 资源的代码如下：

```
eos.transaction(tr => {
    tr.delegatebw({
        from: "testtesttest",
        receiver: "testtesttest", //testtesttest账户为自己抵押
```

仅供非商业用途或交流学习使用

```
        stake_net_quantity: "1.0000 DEV",
        stake_cpu_quantity: "1.0000 DEV",
        transfer: 0
    })
})
```

23. 取消抵押（赎回）CPU 和 Net

通过 eosjs 赎回已抵押的 CPU 和 Net 资源的代码如下：

```
eos.transaction(tr => {
    tr.undelegatebw({
        from: "testtesttest",
        receiver: "testtesttest",
        unstake_net_quantity: "0.1000 DEV", //赎回0.1个DEV
        unstake_cpu_quantity: "0.1000 DEV"
    })
})
```

24. 智能合约部署

如果加载的是 wasm 合约，不用使用 binaryen；如果加载 wast 合约，需要安装并使用 binaryen。

```
npm install binaryen@39.0.0
```

并用以下方式导入到 js：

```
binaryen = require('binaryen')
eos = Eos({keyProvider, binaryen})
```

部署合约的流程以官方自带的 hello 合约为例进行介绍：

```
fs = require('fs')
wasm = fs.readFileSync(`contracts/hello/hello.wasm`) //这个文件在客户端上，而不
    是在服务器上
abi = fs.readFileSync(`contracts/hello/hello.abi`)

eos.setcode('contract_name', 0, 0, wasm) // contract_name 为合约名
eos.setabi('contract_name', JSON.parse(abi)) // @returns {Promise}
```

25. 智能合约的执行

对智能合约进行调用的方法如下。

方法一：通过 .contract 函数完成调用。

仅供非商业用途或交流学习使用

```
eos.contract("contract_name").then(hello => {   //hello随便起的变量名
    hello.hi('axay', {                          //hi是方法名, 'axay'是该hello合
约hi方法的参数
            authorization: ['testtesttest']          //testtesttest是建立该合约的
用户
    }).then(result => {
        console.log(result);
    });
});
```

方法二：通过 transaction 函数完成调用。

```
eos.transaction(
    {
        actions: [
            {
                account: 'contract_name',  //合约名
                name: 'hi',                //方法名, 该方法在官方的hello合约中有
                authorization: [{
                    actor: 'testtesttest',
                    permission: 'active'
                }],
                data: {
                    user: 'axay'
                }
            }
        ]
    }
    // options -- example: {broadcast: false}
).then(result => console.log(result))
```

6.5　DApp 开发实例——网页发币

下面我们将手把手带你开发一个 DApp 的网页发币模块，包括前后端整个流程。在学习之前你应该具有以下基础：html5、css、JavaScript、react。

1. 创建钱包

为了构建实际与钱包和账户交互的 DApp，我们需要设置它们。在终端中输入 cleos wallet create。

这将返回密码，将其保存在安全的地方，因为你很快就会再次需要它。

仅供非商业用途或交流学习使用

注意：本教程采用 docker 运行 EOS 节点程序，如果上面的指令不起作用，则表示你没有在上面的步骤 1 中为“cleos”添加别名。请务必遵循 EOSIO 提供的整个 docker-quickstart 指南。

如果你遇到钱包被锁定的问题，请输入 cleos wallet unlock，然后输入你之前保存的密码。

2. 创建账户

为了与 DApps 互动，你需要在钱包内设置一个账户。我们首先将两组密钥导入我们的钱包，一个是 OwnerKey，一个是 ActiveKey，确保保存两个密钥对。OwnerKey 应始终保持高度安全。

1）首先创建你的 OwnerKey

在你的终端中，键入 cleos create key，终端将返回私钥和公钥。接下来，输入 cleos wallet import（上面创建的私钥），终端将返回导入的公钥值。

2）创建 ActiveKey（与创建 OwnerKey 的步骤相同）

在你的终端中，键入 cleos create key，终端将返回一组新的私钥和公钥。接下来，输入 cleos wallet import（上面创建的私钥），终端将返回导入的公钥值。

3）创建你的第一个账户

在你的终端中键入如下命令。

```
cleos create account eosio user1 [Owner Public Key] [Active Public Key]
```

创建完第一个账户后，你可以按照相同的步骤创建第二个账户（例如，如果你的应用程序让用户将代币发送给另一个账户），那么我就假设你创建了一个名为 user2 的第二个账户。

4）创建 Token

我们下面将创建一种新的 Token。与以太坊不同，使用 EOS 创建 Token 生态系统非常简单，可以从终端完成。我们将部署 EOS 附带的标准代币，但你可以轻松地修改和自定义 eosio.Token 合约，或从头开始构建自己的代币合约。

首先，我们将创建部署代币合约的账户。按照创建账户中的步骤生成 OwnerKey 和 ActiveKey。

然后，在你的终端输入如下命令：

仅供非商业用途或交流学习使用

```
cleos create account eosio eosio.Token [New Owner public key] [New Active public
    key]
```

接下来，我们将标准的代币合约部署到这个新创建的账户中。

```
cleos set contract eosio.Token build/contracts/eosio.Token -p eosio.Token
```

请注意，在上面的行中，-p eosio.Token 表示账户 eosio.Token 授予部署合同的权限。你的钱包可能会被锁定，在这种情况下，你需要按照创建钱包中的解锁步骤解锁钱包。当你创建自己的自定义合同时，你将按照我们在此处所做的方式部署它们，但 cleos set contract eosio.Token 你将替换 eosio.Token 为自定义合同。

接下来，我们将使用刚刚创建的代币合约创建一种新的 Token。你可以根据需要为 Token 命名，并设置发出的 Token 总数。

```
cleos push action eosio.Token create '[ "eosio", "1000000000.0000 TOK" ]'
    -p eosio.Token
```

上面的示例创建了以“ eosio”作为发行者的 10 亿总量的 TOK 代币。最后一步是向我们的一个账户发放代币。请注意，“ memo”是一种将字符串描述附加到事务的方法。此外，我们现在从 eosio 而不是 eosio.Token 获得许可（通过 -p），因为 eosio 账户拥有所有 10 亿个 Token。

```
cleos push action eosio.Token issue '[ "user1", "100.0000 TOK", "memo" ]' -p eosio
```

5）使用 eosjs 将 eos 连接到你的 DApp

接下来将向你展示如何利用 react 实现前端与智能合约之间的交互。

如果你尚未设置 react 应用程序，请键入：

```
create-react-app [name of your react app]
```

在该生成的初始目录中打开终端，在终端中键入以下内容以安装 eosjs 并将其添加到 package.json 中。

```
npm install eosjs -s
```

然后在你的 App.js 文件中，你将要使用以下命令导入 eosjs：

```
import Eos from "eosjs";
```

我们必须初始化一个 eosjs 实例才能在我们的应用程序中使用它。通过将以下内容添加到全局范围来执行此操作。需要注意的是 user1ActivePrivKey 和 user2ActivePrivKey

仅供非商业用途或交流学习使用

是我们在创建账户创建的私有密钥，注意尽量不要在这里使用 Owner 密钥；使用 Active 密钥更安全。

```
const eos = Eos({
    keyProvider: [
        user1ActivePrivKey,
        user2ActivePrivKey
    ]
});
```

现在，我们可以轻松地与本地 EOS 节点进行交互。我们的第一步是检查两个用户账户的代币余额。

```
eos.getCurrencyBalance( "eosio.Token" , 'user1').then(TokenBalance =>      {
    console.log(TokenBalance)
});
```

由于我们在上一节中向用户 1 发出了代币，因此你应该看到当前代币余额为 100.0000 TOK。如果我们检查 user2 的代币余额，你应该看不到任何余额。

```
eos.getCurrencyBalance( "eosio.Token" , 'user2').then(TokenBalance =>      {
    console.log(TokenBalance)
});
```

在这个示例 DApp 中，我们使用了一些非常基本的功能。我们假设 user1 正在接收一些值，并且作为交换，将代币发送给 user2。在 eosjs 中，最好的方法是通过“交易”。在这里，我们将发送 5 个 TOK。

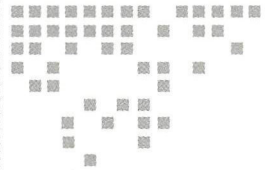
```
eos
    .transaction ({
        actions: [
            {
                account: "eosio.Token" ,
                name: "transfer" ,
                authorization: [
                    {
                        actor: "user1" ,
                        permission: "active"
                    }
                ],
                data: {
                    from: "user1" ,
                    to: "user2" ,
```

仅供非商业用途或交流学习使用

```
                quantity: `5.0000 TOK`,
                memo: "包含交易的一些描述"
            }
        }
    ]
}).then
(result=> {
    //再次检查代币余额。
    //账户中看到5 TOK, 在user1的账户中看到95 TOK
});
```

最后在前端 html 页面中调用这些函数，就可以完成智能合约的交互。

仅供非商业用途或交流学习使用



Chapter 7 第 7 章

基于 EOS 的侧链 / 竞争链开发

在本章中我们将详细阐述 EOS 主链、侧链与竞争链的概念，并解释 BM 提倡的 EOS 生态是如何通过这种多链并存的形式实现的，最后我们将介绍并分析几个当前比较优秀的竞争链案例，帮助你理清侧链 / 竞争链开发的思路。

7.1 侧链 / 竞争链的概念与起源

说起侧链与竞争链的起源，就不得不提比特币。我们都知道比特币系统的目标相对简单：它是一个采用 PoW 共识机制的支持单一原生数字资产传输的区块链，不能与其他任何资产相兑换。这种设计可以简化很多具体实现的复杂度，但随着人们对这个系统的关注度持续上升，人们发现比特币系统这样的设计还是存在诸如吞吐量过低、功能过于单一等一系列问题。所以当时的开发团队们围绕以下几个方面对比特币系统提出优化方案。

- 在可扩展性和去中心化间的取舍。例如，更大的区块可以支持更高的交易速率，代价是加大了验证方的工作量——中心化的风险。
- 在区块链的功能上存在更多的取舍。例如，是让比特币的脚本更强大来支持简明和有用的合约，还是直接减少功能以提升系统响应速度？

仅供非商业用途或交流学习使用

- 一些非货币的资产也可以在区块链上交易，例如欠条、其他合约和智能财产。
- 存在一元化开发的风险：比特币系统由许多密码学组件构成，其中的任何一个组件出问题，都会导致整体价值的损失。如果有可能，不要用同一套算法来保护每个比特币。
- 新技术可能会带来比特币系统创建初期没有想到的新功能。例如，用加密累加器、环签名或 Chaumian 盲签名来增强隐私保护和抗审查性。
- 即使有这样做的迫切需求，但没有对比特币系统安全升级的途径，这意味着任何更改都需要所有参与者协同执行才能生效。比特币的开发者们之间达成了一个共识，对比特币系统的变更必须缓慢、谨慎，且仅当社区有明确意见时才可进行。

然而若仅仅是对比特币系统的升级，则必须要全体维护节点（矿工）达成升级共识，接受并采纳后方能生效，这极大地限制了系统的可扩展性与灵活性。影响力小的团队的优秀想法并不能被采纳，因为他们缺少广泛的共识。

早期针对比特币系统中这些问题的解决方案是可替代的区块链的开发，又称竞争链（altchains）。竞争链除针对上述关注点有所修改外，共用了比特币系统的代码库。之所以被称为竞争链，主要是因为这条新链与作为主链的比特币是存在竞争关系的，两者代码层次极其相似，所以属于有我没他的竞争关系，数字货币用户可以根据自己的判断自主选择使用哪一个系统。

大量竞争链的出现对于区块链技术的发展是存在一定益处的，比如极大地增加了区块链技术的曝光度，让越来越多的人参与进来，也增加了数字货币市场的丰富度，为用户提供选择空间，然而，靠这种创建一个独立的但本质类似的系统来实现技术变更是存在问题的。

- 基础架构碎片化：由于每个竞争链都使用自己的技术堆叠，经常会有重复性或遗失的工作。因此，由于竞争链的实现，可能并未扫除比特币系统中安全特定领域知识的障碍，安全问题常常跨竞争链被复制，而它们的修复也不解决这些问题。必须消耗大量资源或是建立专家评价评审新的分布式加密系统，如果没这样做，安全隐患常常无法察觉，直到漏洞被利用。结果是，我们看到了一种动荡的、无法前行的发展环境，大部分显现出来的项目可能在技术上是最不健全的。打个比方，想象一下在互联网上，每个网站使用自己的 TCP 实现将自

仅供非商业用途或交流学习使用

定义的检验和数据包拼接算法广播给最终用户。这不会是一个能生存的环境，这也不是竞争链能生存的环境。

- 这样的和比特币系统一样，通常有自己的带有浮动价格的原生加密货币，或称竞争币。为了使用竞争链，用户需要通过一个市场来获取这种货币，这将使他们直接面对新货币带来的高风险和波动。此外，因为有独立解决初始分配和估值问题的需求，与此同时还要在不良的网络效应与拥挤的市场中竞争，这既阻碍了技术创新，又助长了市场的博弈。这危及的不仅仅是这些系统的直接参与者，还危及到整个加密货币行业。如果这一领域被公众认为风险过高，对加密货币的采纳可能会受阻，或者有可能完全被遗弃（自愿或立法上）。

这两个问题表明需要有一个环境来让可互操作的竞争链易于创建和使用，但不要产生不必要的市场和开发碎片，即可以在技术上支持区块链间资产的移动，可以通过简单地复用现有比特币的方式，开发出用户可能会采纳的新系统。这也是侧链技术发展的缘由。

这种可互操作的区块链被称为“楔入式侧链”，也就是我们经常说的侧链。简单地说，侧链就像是一条条通路，将不同的区块链互相连接在一起，以实现区块链的扩展。侧链完全独立于比特币区块链，但是这两个账本之间能够“互相操作”，实现交互。一条标准的侧链至少需要具有以下几点特征：

- 在侧链间移动的资产应当能够被当前持有者移回，但除此之外的任何人则不行（包括前持有者）。
- 资产的移动应当无交易对手风险；也就是说，不诚实的一方无法阻止转移的发生。
- 资产转移应当是元操作（原子操作），即要么完全完成，要么根本不发生。不应存在会导致资产损失或允许欺诈产生的失败模式。
- 侧链应当设有防火墙：一个会使某条链发生资产铸造（或偷盗）的缺陷（bug），不应导致任何其他链出现资产的铸造或偷盗。
- 区块链重组时应当处理干净，即使在资产转移的期间也要如此；任何破坏应当只发生在它所处的侧链上。总体来说，理想情况下，侧链应当完全独立，自其他链上的全部所需数据由用户提供。侧链的验证者应当只有在侧链本身的显式共识规则有要求时，才需要跟踪其他链。
- 不应要求用户去跟踪他们未主动使用的侧链。

仅供非商业用途或交流学习使用

侧链的技术方案通常是由资产转移的交易本身提供所有者证明来转移资产，避免让节点有跟踪发送方链的需求。从上层实现角度，当资产从一个区块链向另一个链移动时，我们在第一个区块链上创建交易锁定资产，然后在第二个区块链上创建一笔交易，该交易的输入中包含一个锁定已正确完成的密码学证明。这些输入用某种资产类型来标记，比如创生出资产的区块链的创世哈希 (genesis hash)。

其中第一个区块链就是父链，第二个则简单称为侧链。在某些模型中，两条链可对称地来处理，因此这一术语应当看成是相对的。当资产从（初始）父链转移到一条侧链，可能会再转移到别的侧链，最终还能转回至父链，并保全初始资产。一般大多数系统的父链均是比特币系统，侧链是其他区块链中的某一个。当然，侧链的币 (coin) 也可以在侧链间传递，并非只能与比特币系统进行往来。不过，由于任何一个最初从比特币系统移动的币都可以移回去，所以不管变成什么样，它仍是个比特币。

这样做很好地解决了前面提到的碎片化问题，对只想专注于技术创新的开发者来说是个好消息。

此外，由于侧链是从父链中转移现有资产而不是另外发行新资产，侧链不会引起未经授权的铸币，维护资产的安全和稀缺性依靠父链来实现。

更进一步说，参与者不必再担心他们的持有物会被一个实验性测试版竞争链锁住，因为侧链币能够用一定比例的父链币来赎回。这提供了一个退出机制，减少因软件无人维护而造成的损失。

这一基础架构的额外好处是使父链对本身进行改变不再那么迫切：与其策划一个需要各方均达成一致并一起实施的分叉，不如用侧链来创建一个新的“改变了的比特币系统”。从中期看，如果广泛认可新系统有进步，最终会明显看到，相对于比特币父链，更多的人会使用这个新系统。由于父链的规则没有任何修改，任何人都可以根据自己的时间去切换，不会有任何因共识失败所带来的风险。从长期看，如果父链被认为或当父链需要改变时，侧链的成功将给父链的更改提供所需的信心。

7.2　侧链实现方式

我们知道侧链实现是通过双向锚定（Two-way Peg）技术，实现暂时的将数字资产在主链中锁定，同时将等价的数字资产在侧链中释放，同样当等价的数字资产在侧

非卖品，仅供非商业用途或交流学习使用

链中被锁定的时候，主链的数字资产也可以被释放。然而双向锚定实现的最大难点是协议改造需兼容现有主链，也就是不能对现有主链的工作造成影响，其具体实现方式可以分为以下几类。

1. 单一托管模式

最简单的实现主链与侧链双向锚定的方法就是通过将数字资产发送到一个主链单一托管方（类似于交易所），当单一托管方收到相关信息后，就在侧链上激活相应数字资产。这个解决方案的最大问题是过于中心化。图 7-1 给出了以比特币为主链的单一托管模式的工作原理示意图。

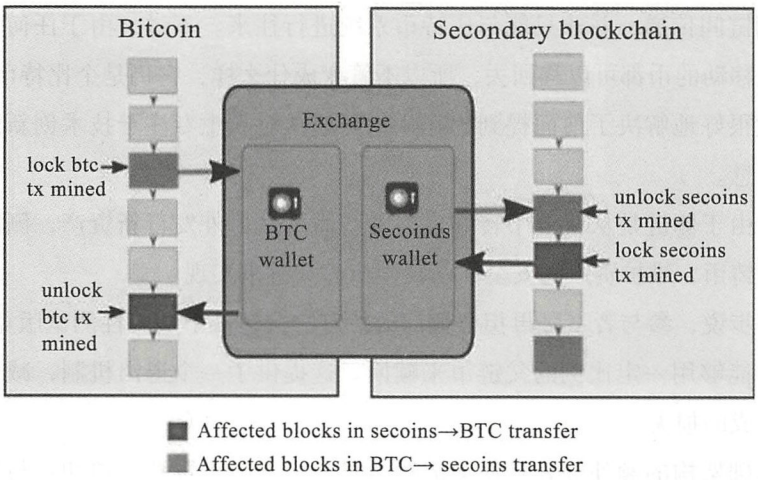


图 7-1 单一托管模式

2. 联盟模式

联盟模式是使用公证人联盟来取代单一的保管方，利用公证人联盟的多重签名对侧链的数字资产流动进行确认。在这种模式中，如果要想盗窃主链上冻结的数字资产就需要突破更多的机构，但是侧链安全仍然取决于公证人联盟的诚实度。图 7-2 给出了以比特币为主链的联盟模式的工作示意图。

单一托管模式与联盟模式的最大优点是它们不需要对现有的比特币协议进行任何改变。

3. SPV 模式

SPV（Simplified Payment Verification）模式是最初的侧链白皮书《Enabling Blockchain Innovations with Pegged Sidechains》中的去中心化双向锚定技术最初设想。SPV 是一种

非卖品，仅供非商业用途或交流学习使用

用于证明交易存在的方法，通过少量数据就可以验证某个特定区块中交易是否存在。在 SPV 模式中，用户在主链上将数字资产发送到主链的一个特殊的地址，这样做会锁定主链的数字资产，该输出仍然会被锁定在可能的竞争期间内，以确认相应的交易已经完成，随后会创建一个 SPV 证明并发送到侧链上。此刻，一个对应的带有 SPV 证明的交易会出现在侧链上，同时验证主链上的数字资产已经被锁住，然后就可以在侧链上打开具有相同价值的另一种数字资产。这种数字资产的使用和改变在稍后会被送回主链。当这种数字资产返回到主链上时，该过程会进行重复。它们被发送到侧链上锁定的输出中，在一定的等待时间后，就可以创建一个 SPV 证明来将其发送回主区块链上，以解锁主链上的数字资产。SPV 模式存在的问题是需要对主链进行软分叉。图 7-3 给出了以比特币主链的 SPV 模式的工作流程示意图。

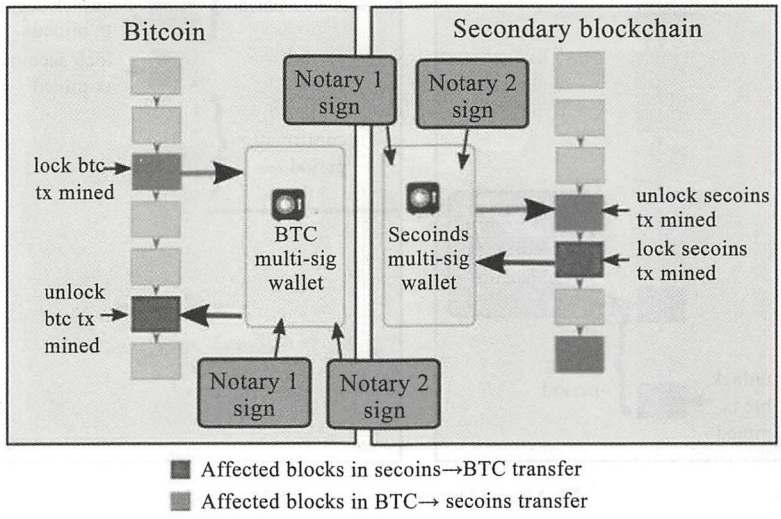


图 7-2　联盟模式

4. 驱动链模式

驱动链概念是由 Bitcoin Hivemind 创始人 Paul Sztorc 提出的。在驱动链中，矿工作为“算法代理监护人”对侧链当前的状态进行检测。换句话说，矿工本质上就是资金托管方，驱动链将被锁定数字资产的监管权发放到数字资产矿工手上，并且允许矿工们投票何时解锁数字资产以及将解锁的数字资产发送到何处。矿工观察侧链的状态，当他们收到来自侧链的要求时，他们会执行协调协议以确保他们对要求的真实性达成

非卖品，仅供非商业用途或交流学习使用

一致。诚实矿工在驱动链中的参与程度越高，整体系统安全性也就越大。如同 SPV 侧链一样，驱动链也需要对主链进行软分叉。图 7-4 给出了以比特币为主链的驱动链模式的工作流程示意图。

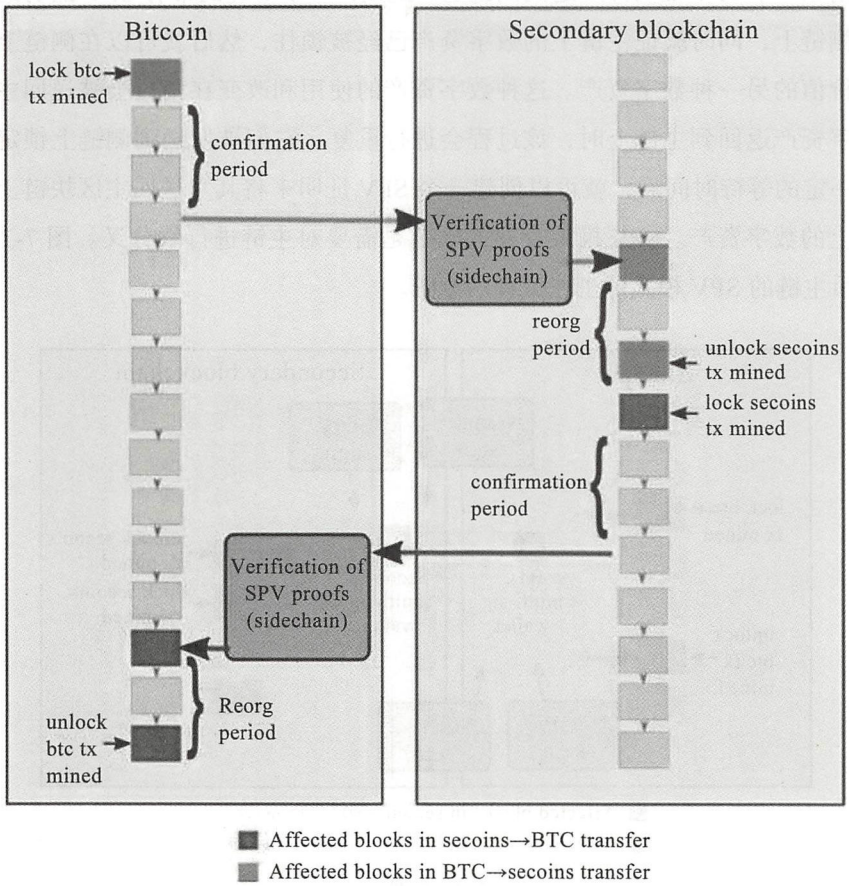


图 7-3 SPV 模式

5. 混合模式

上述所有的模式都是对称的，而混合模式则是将上述获得双向锚定的方法进行有效结合的模式。由于主链与侧链在实现机制上存在本质的不同，所以对称的双向锚定模型可能是不够完善的。混合模式是在主链和侧链使用不同的解锁方法，例如在侧链上使用 SPV 模式，而在主链网络上则使用驱动链模式。同样，混合模式也需要对主链进行软分叉。

非卖品，仅供非商业用途或交流学习使用

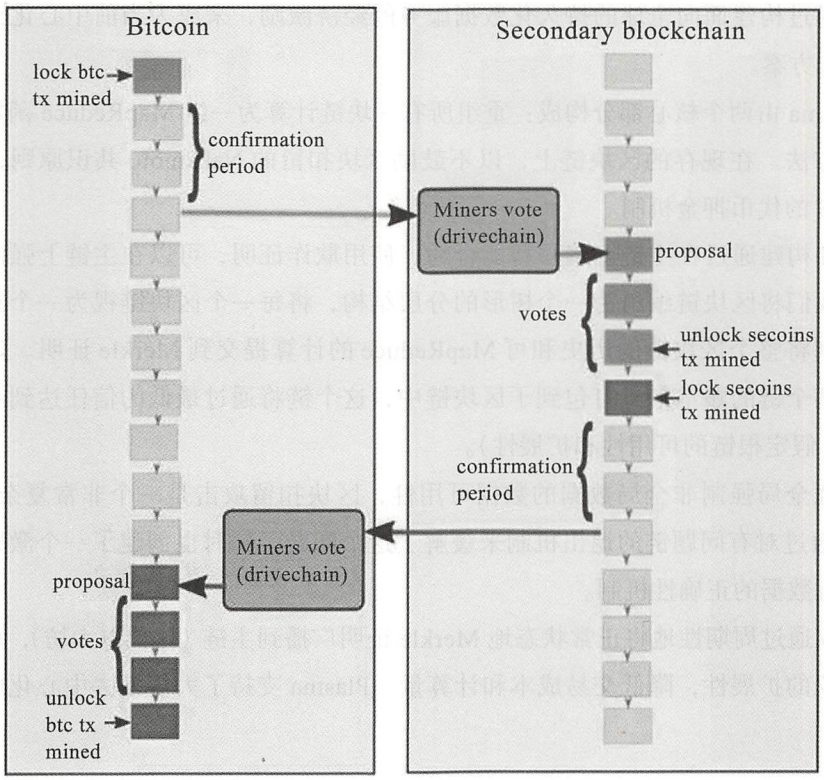


图 7-4　驱动链模式

7.3　主流侧链项目

1. Plasma

以太坊的侧链技术 Plasma，使交易效率大幅提升至每秒几千 TPS，满足大量商业应用需要。Plasma 是一个激励，也是强制智能合约执行的框架。其可以扩容达到每秒大量的状态更新（能达到每秒 10 亿级），在区块链上能支持全球范围内的大量的去中心化金融应用。这些智能合约通过网络交易手续费用于激励持续的自动化运作，最终依赖于底层的区块链（比如以太坊）来强制交易状态的锁定。

Plasma 提出的这种去中化的可扩展自动运行应用，不仅可以用来处理金融行为，

非卖品，仅供非商业用途或交流学习使用

也可以通过构建面向全球的持久化数据服务的经济激励，来成为当前中心化云服务的一种备选方案。

Plasma 由两个核心部分构成：重组所有区块链计算为一组 MapReduce 函数和一个可选的方法。在现存的区块链上，以不鼓励区块扣留的 Nakamoto 共识原则，来实现一个 PoS 的代币押金机制。

这种构建通过在主链上编写智能合约，使用欺诈证明，可以在主链上强制状态的锁定。我们将区块链编组为一个树形的分层结构，将每一个区块链视为一个独立的分支，强制将整个区块链的历史和可 MapReduce 的计算提交到 Merkle 证明。通过主链强制将某个链的账本信息打包到子区块链中，这个链将通过最低的信任达到不可思议的扩容（假定根链的可用性和扩展性）。

围绕全局强制非全局数据的数据可用性，区块扣留攻击是一个非常复杂的问题。Plasms 通过对有问题链的退出机制来缓解了这个问题，同时也创建了一个激励和持续强制执行数据的正确性机制。

仅仅通过周期性地将正常状态地 Merkle 证明广播到主链（比如以太坊），就将带来不可思议的扩展性，降低交易成本和计算量。Plasma 支持了大规模去中心化应用的持续运行。

2. RootStock

RootStock 是一个建立在比特币区块链上的智能合约分布式平台。它的目标是将复杂的智能合约实施为一个侧链，为核心比特币网络增加价值和功能。RootStock 实现了以太坊虚拟机的一个改进版本，它将作为比特币的一个侧链，使用了一种可转换为比特币的代币作为智能合约的“燃料”。

根链和以太坊在功能上是高度重合的，它们都是一个点对点的分布式计算网络。根链没有发行任何代币，它是和比特币一起挖矿的，称为联合挖矿。

Rootstock 采用双向锚定技术，双向锚定技术经常被认为是一种将 BTC 转移到 SBTC（智能比特币，这里我们可以理解为就是现在的侧链）的方法，反之亦然。实际上，当 BTC 被交换成 SBTC，区块链之间并没有货币被“转移”。没有任何一笔交易实施了这个动作。 这是因为比特币不能验证另一条区块链上的余额属性。当用户打算把 BTC 转换成 SBTC，把部分比特币在锁定比特币区块链上，同时在 RSK 上释放等量的 SBTC。当 SBTC 需要换回比特币时，再次在 RSK 上锁定 SBTC，同时在比特币区块链上释放等量的比特币。通过安全协议保证相同的比特币不会在两条区块链上同时释放。

非卖品，仅供非商业用途或交流学习使用

当比特币用户想要使用双向锚定时，会发送一笔交易到由联盟保护资金的多重签名钱包上。RSK 区块链使用这笔交易中比特币相关的公钥来存储智能比特币（SBTC）。这意味着比特币区块链上控制比特币的私钥可以用来控制 RSK 链上的账户。尽管公钥和私钥是相似的，但是每个区块链使用不同的格式编码地址。这意味着，两个区块链上的地址是不同的。

RSK 像比特币一样，由基于 SHA256D 算法的工作证明保护。如果所有的 RSK 矿工串通，他们可以审查一笔或所有的 RSK 交易，但是他们不能偷走智能比特币（SBTC）或者比特币。此外，RSK 矿工不能双花，因为联邦提供检查点服务，并且所有的联邦成员节点将被高度链接到 RSK 网络，防止 Sybil 攻击。联邦将使用检查点功能来防止与协议故障无关的高深度重组。 联邦不能双花，因为联邦成员不允许检查点具有冲突交易的两个区块。

RSK 平台的安全性取决于参与联合挖矿的数量和联邦成员的数量和质量（安全合格）。超过 30% 的比特币矿工已经在与 RSK 合作，剩下的 50% 计划加入 RSK 挖矿，因此我们期望达到超过 51% 的比特币矿工来保证 RSK 网络的安全。因为联邦添加了第二层防护，从共识的角度来说，我们认为 RSK 平台比比特币更安全。

3. Liquid

Liquid 是 Blockstream 的第一个商业侧链。它可以在交易所之间实现资金的即时移动，而无须等待比特币区块链中的确认延迟。它可供参与比特币交易的用户使用。

4. POA

POA 网络是一个开放的以太坊侧链，具有 PoS 权威证明共识，由独立验证者达成。POA 正在为智能合约建立一个公共网络，它具有速度快、安全性高和成本效率高的优点。

POA 具有以下几个特征。

- 5 秒阻塞时间：网络与以太坊协议完全兼容。然而，PoA 共识机制允许更快的阻塞时间。
- 无挖矿机制：在 PoA 网络中创建区块不需要大量的计算资源，如基于工作证明的区块链。
- 内置治理模式：验证者投票使用 Governance DApp 添加或删除验证器。因此，网络支持在共识级别进行投票。

非卖品，仅供非商业用途或交流学习使用

- 法律上可识别的硬分叉决策：所有验证者都必须与网络签署法律协议。硬分叉决策具有法律约束力。
- 混合网络模型：PoA 网络模型可以作为蓝图快速启动具有内在价值的开放式内部部署网络。
- 支持的 DApps：网络支持广泛的“身份证明”DApps，并将它们集成在一起，以证明验证者的身份。

7.4 EOS 侧链

首先我们明确一下当前 EOS 主链、侧链与竞争链的概念。目前对 EOS 主网的主要共识应该是 2018 年 6 月初启动的全球性 EOS 网络，基于 EOSIO 开发的区块链项目中承认主网数据并可以与主网做信息交互的被称为 EOS 侧链，其余的项目为 EOS 竞争链。这些竞争链有自己独立的 Token、超级节点，并且不可以与主链做信息交互。那么 BM 为什么要做这样的设计呢？

我们知道 EOS 的目标是达到百万级 TPS，然而当前经过压力测试，主网中最高 TPS 为 3000 左右，产生这样大差距的主要原因有二，其一是目前的主网依然运行的是单线程版本，相比于白皮书中的多线程方案肯定是存在差距的，但是可预期的是，Block.one 一定会向着这个目标进发；其二就是目前还没有比较成功的 EOS 侧链出现，而通过无限扩容的 EOS 侧链群所建立的 EOS 生态可以显著提升 EOS 的 tps，究其原因就相当于为主链做了负载均衡，分摊主链交易量，进而提升 EOS 生态的总 TPS，这样的架构如图 7-5 所示。

那么 EOS 是如何实现主链与众多侧链的信息交互呢？主要通过跨链通信机制实现的，EOS 系统软件被设计为跨区块链通信友好的。 这是通过生成操作存在证明与操作时序证明变得简单而实现的。 这些证明与应用架构设计相结合，即围绕操作细节的跨链传输和有效性验证时隐藏应用程序开发者的架构设计。如图 7-6 所示是基于 MerkleTree 的跨链机制示意图。

如果客户端不需要处理所有的交易会让多区块链间的整合更为轻松。毕竟，一个传统的中心化交易所只需要关心交易所的入账和出账，别无他求。如果交易所链条可以使用资金的轻量 merkle 证明，而不必非要完全依赖对它区块生产者的信任会

非卖品，仅供非商业用途或交流学习使用

是一个不错的主意。至少一个链的区块生产者在与其他区块链同步时更乐意保持尽可能小的开销。LCV 的目标能产生相对轻量存在性证明，使得任何追踪相对轻量数据集的人可以验证其有效性。

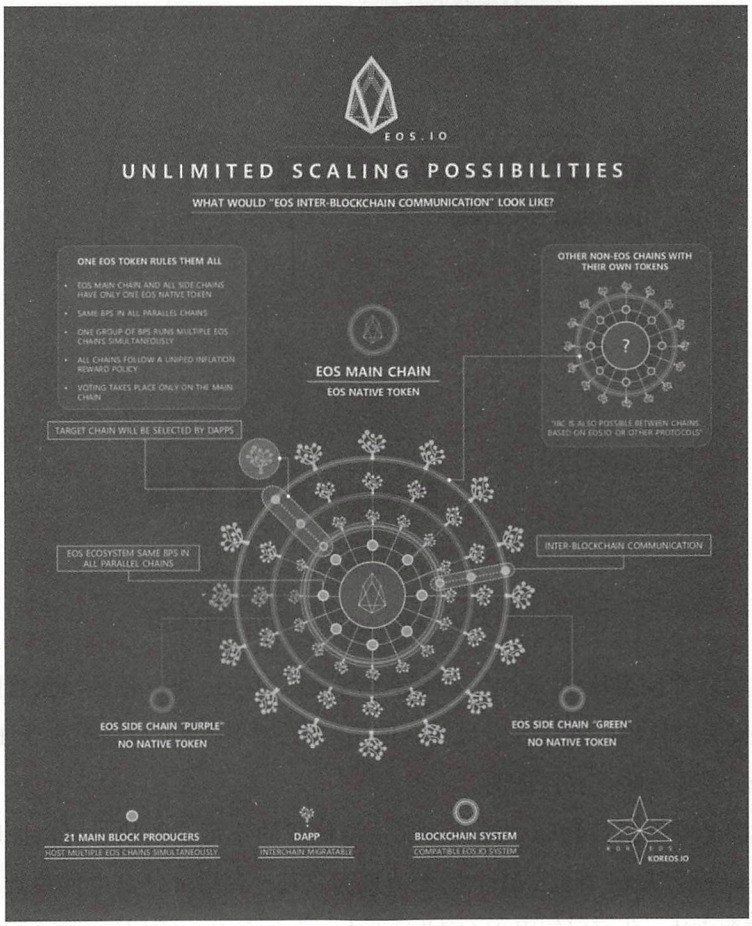


图 7-5　EOS 生态架构图

在这种情况下，目的是为了证明一个特定的交易是包含在一个特定的区块中，区块包含在一个特定的区块链的已验证历史中。

比特币支持通过全节点的完整记录获取每年 4MB 大小的区块头信息来验证交易。每秒 10 个交钟，一个有效的证明需要 512 个字节。这对于有 10 分钟间隔的区块链没有问题，但是对于 0.5 秒间隔区块链就显得不那么“轻量”了。

非卖品，仅供非商业用途或交流学习使用

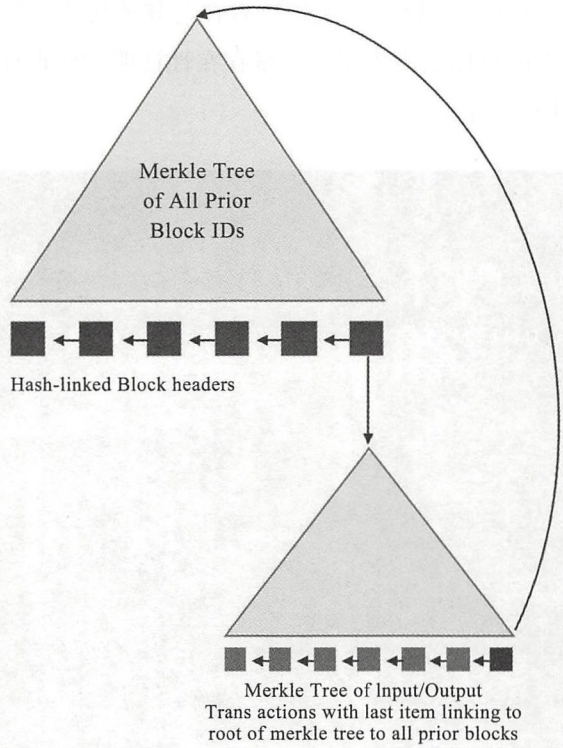


图 7-6 基于 MerkleTree 的跨链机制示意图

EOS 系统软件使得任何一个人在拥有了与交易对应区块，且区块包含随意一个不可逆的区块头后，就可以进行轻量证明。使用哈希链结构就可以通过少于 1024 字节的空间来完成任意交易的存在性证明。

给定区块链中的任何块 id，并且头部是可信任的不可逆的块。可以证明块包含在区块中。这个证明以 ceil (log2(n)) 个摘要消息为路径，其中 n 是链中的块数。给定一个 SHA256 的摘要方法，你可以使用 864 字节证明在一个包含 100 万块的区块链中存在任何块。

使用正确的哈希链接生成块来实现这些证明，这意味着没有理由不以这种方式生成块。

当需要验证其他链时，有譬如时间 / 空间 / 带宽的多样化优化可以做。追踪全部区块头（420 MB/ 年）将保持证明体积的轻巧。只追踪 Y 近的头可以提供 Y 小长期存储和证明大小来获得。另外，一个区块链可以使用懒惰的评估方法，即它记住过去证明的中间值哈希。新证明只需要包含指向已知稀疏树的链接。确切的方法将取决于那些包含对 Merkle 证明引用的交易所在的外部区块的比例。一定密

仅供非商业用途或交流学习使用

度的联系后，将变得更为高效，一个链会包含另一个链整个区块的历史和消除证据一起，这样就不需要通信便可以验证了。

出于性能原因，应最小化提交跨链证明的频率。考虑到跨链通信的延时，EOSIO 与外部区块链进行通信时，区块生产者必须等待直到 100% 确信一个交易已经被另一个区块链确认为不可逆后才会接收它成为一个有效的输入。因为如果任何一个区块生产者不需要等待区块成为不可逆转状态时就做跨链动作，就会遭遇类似一个交易所签署了一笔存款后来被逆转了的问题，可能影响到区块链的共识的有效性。EOS 系统使用 DPoS 和 aBFT 来提供快速的不可逆转性，使每个块的确认时间降低到 0.5 秒。

当使用来自外部区块链的 Merkle 证明时，在已知所有交均已验证和已知没有交易被跳过或遗忘之间有一个重要的差异。虽然不可能证明所有 Y 近的交易是已知的，但有没有间隙的交易历史是可以被证明的。EOS 系统软件在每个用户的每个传递的消息上分配了一个序列号。一个用于可以使用这些序列号来证明所有的消息由某个特定账户处理，只需要看它是否是按序执行的。

EOS 系统同样计划隔离见证人机制。隔离见证人（SegWit）的概念是，交易签名在交易可改变地包含在区块链中之后就是无关的了。一旦它是不可变的，签名数据可以被删除掉，其他人仍然可以获得当前状态。 由于签名在大多数交易中占很大比例，因此 SegWit 节省了大量的磁盘使用和同步时间。

这个概念同样适用于跨链通信的 merkle proofs。证明一旦被接收并且不可逆转地记录到区块链中，则证明使用 2KB 的 SHA256 哈希就不再是获得区块链状态所必需的了。在跨链通信的情况下，这种处理方式比普通签名可节省更多空间，其所需空间仅为普通签名的 1/32。

7.5　EOS 竞争链

本节我们将介绍几种比较优秀的 EOS 竞争链，为大家提供设计 EOS 竞争链的灵感。客观地说，自 EOS 主网发布以来的这一路并不平坦。虽然许多人认为，随着主网的发展，出现问题是不可避免的，但是当前问题其实并不仅仅是技术问题。总结下来主要有以下几点问题：

- 缺乏功能治理，无法追究不良行为者的责任；

仅供非商业用途或交流学习使用

- EOS 的 Token 分配不平衡；
- 没有支持账户冻结的监督机制；
- RAM 价格极不稳定，DApp 开发者成本过高。

这些都极大地阻碍了 EOS 当初取代以太坊的宏大目标。当前面临的这些问题也成为众多竞争链的主要优化方向。当前 EOS 竞争链大多仅仅是增加了更多的空投福利，我们在这不做介绍，所以我们仅以 Telos 为例，给大家一个对上述问题的解决思路。

Telos 是一条 EOS 的竞争链，它的开发团队认为当前 EOS 主网面临的主要问题是巨头持币者权力过大，EOS 的前十名持有者拥有近一半的所有代币，这已不是什么秘密。如果持有 Token 没有带来一定的影响力（1 单位 Token= 1 票）的话，这些巨头持币者就不会成为问题。虽然根据目前的 EOS 宪法，投票购买是非法的，然而目前一些超级节点确实是严重依赖支持它们的一些巨头持币者的投票来“掌权”（并获得块奖励以资助他们的运营）。所以由于目前没有监督机制可以让这些不良行为者承担责任，社区便无力阻止这种情况发生。

针对这些问题，Telos 通过推出了一个基于 EOSIO 代码库（“分叉”）的竞争链来解决这个问题。在 Telos 中，TLOS 代币（Telos 的代币名称）将会根据 EOS 主网中某一时刻的 snapshot，以 1:1 的比例分配给每个持币者，每个账户的分配上限为 40000。这种方法有以下两个优点：

- 它消除了网络中的巨头持币者数量，从而将权力重新公平地分配给社区。
- 它显著减少了 Token 供应数量，造成了更多的稀缺性和的更高的潜在估值。

根据 Telos 白皮书，Telos 代币供应将是 EOS 的五分之一到三分之一，这将使其总量在 3.3 亿的范围内。如图 7-7 所示，从 Etherscan 中的数据我们可以看到，在主网上线之前，持有基于 ERC-20 的 EOS 代币的前 500 个以太坊地址，它们共同拥有大约 8.3 亿个 EOS。若按照 Telos 的方法分配代币，所有持币者的分配上限为 4 万 Telos，这意味着 8.3 亿票变为 2000 万，极大地降低了巨头持币者的权力。

Telos 的另一个好处是它对投票机制中增加的反向加权机制，即对于在投票过程中不使用全部 30 票的代币持有者（代币持有者可以将其代币的一次投票投入最多 30 个 BP），他们的投票权重将会降低，这将会导致总票数分布更均匀，使投票结果更加公平，其投票权重计算方式如图 7-8 所示。

针对超级节点，与 EOS 一样，Telos 的 21 个超级节点都获得相同数量的出块奖励。

仅供非商业用途或交流学习使用

不同点在于，Telos 中的备选区块生产者实际上需要定期生成块，而不仅仅是当前只有在 21 个超级节点中的任何一个节点都没有响应时才生产。这样，Telos 确保所有区块生产者随时都可以生产，这降低了网络中断的风险。

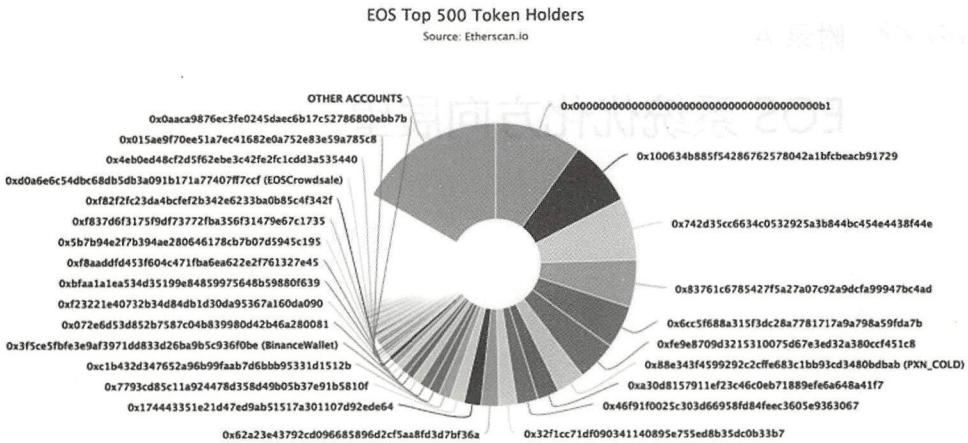


图 7-7　EOS 代币持有情况

$$Normal\ Voting \cdot \sin\left(\frac{\pi}{2} \cdot \frac{number\ of\ votes\ cast}{maximum\ number\ of\ votes\ possible}\right)$$

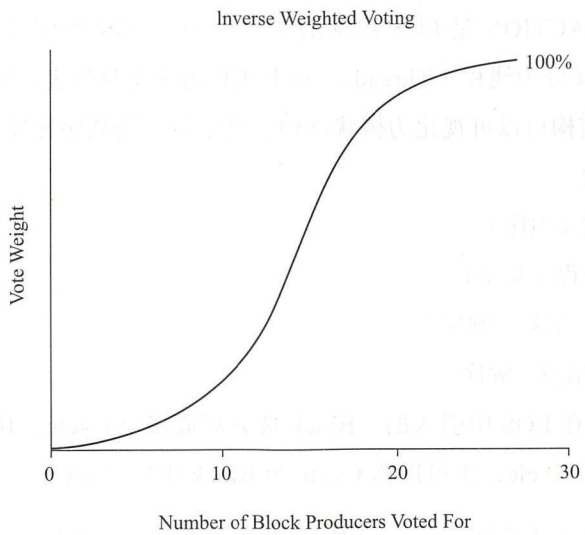
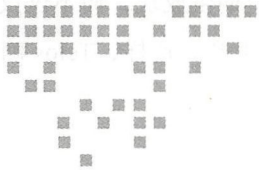


图 7-8　Telos 投票权重计算

仅供非商业用途或交流学习使用



Appendix A 附录 A

EOS 系统优化方向展望

A.1 百万 TPS 的实现目标

1. action 异步执行

多核异步执行 ACTION 是 EOS 系统的一大亮点。EOS 系统将每个块分为周期（Cycle），每个周期又分为线程（Thread），每个线程包含交易列表，每个交易包含一组要传递的消息。该结构可以可视化为树状结构，其中各层按顺序并发处理。

- Block：区块
- Cycle：周期（顺序）
- Threads：线程（并发）
- Transactins：交易（顺序）
- Messages：信息（顺序）

Cycle 的概念是在 EOS 中引入的。Block 被分割成多个 Cycle，块生产者可以在它的当前块中生产多个 Cycle，也可以称 Cycle 为 Block 中的“小链”。

```
Block[Cycle,Cycle,Cycle...] <--- Block[Cycle, Cycle,Cycle...]
```

这样做的好处是：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 提升了消息交互的效率，后面的 Cycle 可以依赖前面 Cycle 的数据，而不需要等完整的 block（Cycle 生成完会立刻异步广播出去）。
- 使网络资源的使用更平滑，降低传输毛刺，使得块生产者的交接开销在 1～2 TCP RTT，无论块有多大。

引入 Threads 的概念，主要是为了并行执行。BP 在打包 Cycle 的时候，可以利用多 core 资源进行并行打包，相关的交易 / 消息调度路由到同一个 Thread，由单独的 core 处理，Thread 与 Thread 之间的数据不共享，EOS 主网刚上的时候是所有的交易由单个 core 进行打包。

EOS 并行化的挑战在于对多个 core/thread 进行 context 分片后，无法做到不依赖全局统计数据。

在并行计算的时候为了保证性能，不会实时全局统计，必须要以一种相对大粒度的方式来更新统计数据，如 Block 粒度或 Cycle 粒度。事实上 EOS 是以 Cycle 维度，先打包交易，Cycle 处理完发送前进行 limit 统计校验，如果有账户超过了限制，则通过修正的方式（类似于 branch miss-prediction）来 UNDO 这些交易，分支预测失败是会明显损失性能的，那么如何抵御此类攻击？

EOS 主网目前采用的是单核心打包交易，目前的 TPS 达到了 3000 多，随着后续 EOS 主网的多核心部署，EOS 会向着百万级 TPS 目标前进。

2. 竞争链与侧链带来的负载均衡

侧链是一种新兴机制，允许在一个区块链中的代币和其他数字资产被分离到另一个区块链中安全地使用，然后在需要时将它们转移回原始的区块链。侧链是连接到另一个区块链（通常称为主链）上的单独区块链。它们使用双向挂钩相互连接，这意味着主链和侧链上的资产可以以预定的利率进行交易。为了实现主链与侧链之间进行的资产交易，首先需要将它们在主链上的资产发送到某个地址，从而有效地锁定资产。交易完成后，将向侧链传达确认信息。然后，侧链会将侧链上的一定数量的资产释放给用户，相当于主链上“锁定”的资产数量乘以汇率。反之亦然，为了将侧链中的资产换成主链中资产，人们需要做类似的事情。

侧链需要它们自己的矿工 / 出块节点，因此每个侧链都是独立的。如果侧链出现故障或受到损害，主链仍可继续运行。如果主链被破坏，侧链仍然可以继续运行，尽管第二种情况对侧链来说很可能是灾难性的。因此侧链允许新的和可能不稳定的软件

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

在侧链上部署和测试。如果软件对区块链造成损害，则该损坏只包含在侧链内。

另一方面，EOS 的目标是实现百万级的 TPS，从技术上来说，在较短时间内，即便是多线程的版本也不可能实现百万，只能通过平行链或者相对独立的子链完成。这些链能跟 EOS 主链进行直接交互，链和链之间也可以进行直接交互，就像不同银行间能跨行转账一样自然。因此，如果用户或开发人员对主链发送交易的成本和速度不满意，他们可以在其中一个侧链上使用或部署他们的 DApp。这导致更多样化的网络与更强大、更快速且更健壮的主链。

A.2 跨链交互

高效的跨链通信是可扩展性和协议演进的关键。当我们学会如何扩展时，一个代币可以轻松地从一代链转移到下一代。当代链和后代链可以同时并排运行。

EOS 系统中的软件被设计为跨区块链通信友好的，即 EOS 可以支持跨链交互（支付、交易、转账、发消息），这种跨链包括子链和侧链（Side-Chain）。侧链是由 EOS 源代码直接发起的独立公链，类似于一种硬分叉。侧链可以拥有自己的委员会、见证人以及计算资源，有自己的代币，这些代币有自己的增发方式，而且代币可以和 EOS 通过接口进行锚定去实现包括锁定在内的某种操作关联，计算资源提供者也会为侧链提供资源。子链是主链通过逻辑方式形成的，这些子链共享见证人计算资源，通过跨链机制进行交互。

跨链的实现是通过简化生成消息存在的证明和消息序列的证明来实现的。这些证明与围绕消息传递设计的应用程序体系结构结合在一起，使跨链通信交互和验证的细节能够被应用程序开发人员隐藏起来。

1. 用于轻客户端验证的 Merkle 证明（LCV）

EOS 使用“默克尔 (Merkle) 证明”来实现区块链交互，默克尔树（又叫哈希树）的特点是，底层数据的任何变动，都会传递到其父亲节点，一直到“树根”。

默克尔树的典型应用场景包括下列几种。

- 快速比较大量数据：当两个默克尔树根相同时，则意味着代表的数据必然相同。
- 快速定位修改；

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 零知识证明：例如如何证明某个数据（D0，…，D3）中包括给定内容 D0，很简单，构造一个默克尔树，公布 N0、N1、N4、Root，D0 拥有者可以很容易检测 D0 存在，但不知道其他内容。

要更容易地与其他区块链集成，对于客户端而言最好是不处理全部的交易。毕竟，一个交易所只需要关心交易所的入账和出账，其他的就不需要其关心。如果交易所链条可以使用资金的轻量默克尔（Merkle）证明，而不完全依赖对它区块生产者的信任会是一个不错的主意。至少一个链的区块生产者在与其他区块链同步时更乐意保持尽可能小的开销。

LCV 的目标是能产生相对轻量级的交易存在证明，并且该证明能被其他人通过跟踪一个轻量级数据集进行验证。既然如此，目的就是证明一个特定的交易是被一个特定的区块包含其中，并且这个区块是被包含在已经验证的特定区块链历史中。

比特币的轻量级验证方式是，假设所有节点都有读取区块头数据完整记录的能力。而区块头数据每年增长 4MB，假设每秒产生 10 笔交易，一个有效的证明需要 512 bytes，这对于一个出块时间为 10 分钟的区块链来说是可行的。但对于一个出块时间为 3 秒的区块链来说远远不够。

EOS 软件的轻量级证明只需要验证包含某个特定的不可逆交易之后的区块头数据，使用哈希链表架构，数据集可以保持在 1024 bytes 内，即可证明任何一个交易是否存在。这是基于验证节点保留着前一天的所有区块头数据（2 MB 大小），然后证明这些交易只需要 200 bytes 大小的证明数据。

当生成区块使用合适的哈希链表时，使用这种方法只会带来很小的增量开销，这意味着没有理由不以这种方式去生成区块。

当与其他链验证证明的时候，时间、空间和带宽都有很大的优化空间。跟踪所有区块头数据（420MB/ 年）可以使证明体积尽可能小。只跟踪最近的区块头可以使得在持久区块头数据保存体积以及证明体积之间获得平衡。同样的一个区块链可以“懒惰地”只记录过去数据的哈希值作为之前数据的证据，新证明只需要保留已知的 sparse tree（稀疏树）结构，具体的方法会视乎于外部区块所占的默克尔证明所包含的交易比例。

在链与链之间经过一定密度的相互关联之后，它们将会变得越来越高效。一条链可能会包含另外一条链的全部历史记录，那么就不再需要互相证明。从性能的角度来

非卖品，仅供非商业用途或交流学习使用

说，这将极大地减少链间互相证明操作的频率。

2. 跨链通信延迟

与其他区块链通信的时候，矿工必须等待其他区块链不可逆地确认之后才会接受其为有效的输入。使用 EOS 系统软件，凭借出块时间为秒的委任权益证明以及 21 个矿工，这大概只需要 45 秒的确认时间。如果某个链上的矿工不等到交易确认，就像一个交易所接受了一笔存款而后又撤销这笔操作，这会影响这条链共识的有效性。

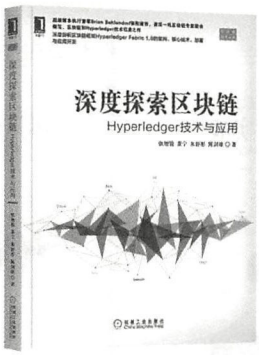
3. 完成证明分发机制

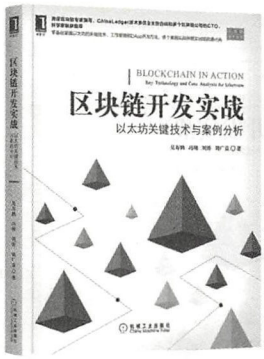
使用外部区块链的默克尔证明时，知道所有已处理的交易是有效的和知道有没有交易被忽略，这两者之间有巨大的不同。因为不可能证明所有最近交易是已知的，但有可能证明历史交易数据之间没有缺失。EOS 操作系统通过分配一个顺序的标识编号给每一笔到达账户的信息来完成这个功能，用户可以使用这些标号来证明所有给这个账号的消息已经被处理并且是被按顺序处理的。

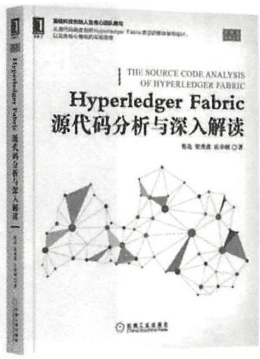
仅供非商业用途或交流学习使用

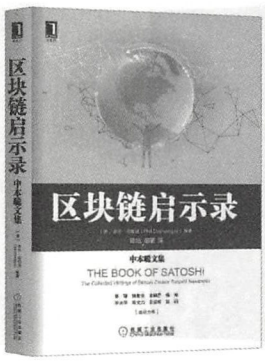
非卖品，仅供非商业用途或交流学习使用

推荐阅读



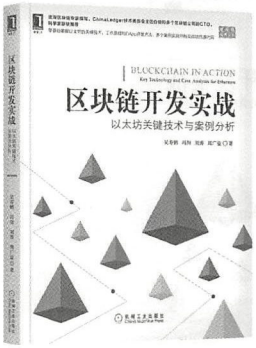


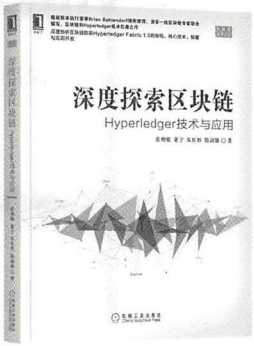


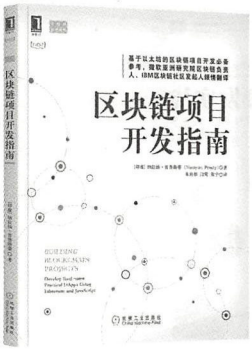


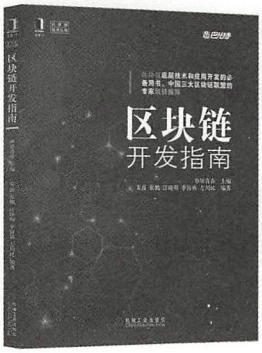
仅供非商业用途或交流学习使用

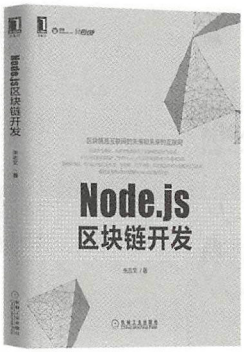
推荐阅读











仅供非商业用途或交流学习使用

作者简介

李万才 复旦大学博士，具有国际影响力的区块链技术专家，EOS 基金会董事，EOS 引力区联合创始人和水滴资本管理合伙人，国家技术转移东部中心区块链产业中心特聘专家，全球石墨烯区块链应用中心理事。

复旦大学计算机科学技术学院外聘教师。长期在科研院所从事技术研发工作，是数据加密、网络安全及物联网技术方面的专家。主持和参与过国家级、省部级科研项目多达数十个，发表过 EI、SCI 论文十多篇，获得十多项专利。

快速投稿通道

联系人：杨福川

邮箱：yfc@hzbook.com

微信：15693352

仅供非商业用途或交流学习使用

EOS 是石墨烯区块链架构的代表作品，也是高性能智能合约平台的领军者。本书为爱好者和开发者提供了珍贵的从入门到精通的全面教程，必然会在 EOS 公链生态的建设中发挥重要作用。

——巨蟹（刘嘉陵） 全球石墨烯区块链应用中心理事长 / 比特股理事

开发者是新世界的创造者。区块链技术本身就源于一群密码学朋克开发者，此后的发展和演进更是离不开前赴后继投身其中的开发者们。希望这本书可以给有意加入这个新世界的开发者们提供帮助，说不定下一个改变世界的程序员就在其中。

——老猫（余文卓） BigONE CEO/ 雄岸基金管理合伙人

李博士是我认识的 EOS 社区生态建设者中为数不多的让我敬佩的人之一。他长久以来以一个 EOS 持有者、学者、师长、兄长的身份活跃在社区建设中。本书更是融合了他对于 EOS 的理解和热爱，内容深入浅出，适合区块链开发者和爱好者充分了解 EOS 的技术和理念。

——曹飞 火币矿池总经理

本书深入浅出地讲解了区块链的运行原理、共识机制、治理模式、智能合约及应用实现等内容，十分值得推荐。

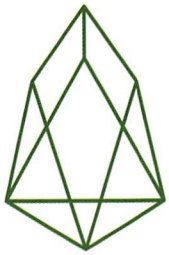
——胖哥（施瑞琦） EOScannon 联合创始人

李博士热爱 EOS 并全身心投入其中，推动着整个 EOS 生态蓬勃发展。本书条理清晰，带领读者深入了解 EOS，从基本常识到共识机制，再到节点治理，环环相扣地介绍了 EOS 的每一个环节和知识点，对开发者会很有帮助。

——李昂 Starteos 首席执行官

EOS 是用户真正可以应用到全行业的区块链技术，互联网通过它实现价值互联网进化。本书系统介绍了 EOS 的技术和原理，每一位开发者都不应错过。

——高锋 EOS MEET.ONE 创始人



投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机\区块链
ISBN 978-7-111-61529-3
9 787111 615293
定价：79.00元